User's Manual

RI850V4 V2

Real-Time Operating System

User's Manual: Coding

Target Device
RH850 Family (RH850G3K)
RH850 Family (RH850G3M)

All information contained in these materials, including products and product specifications, represents information on the product at the time of publication and is subject to change by Renesas Electronics Corp. without notice. Please review the latest information published by Renesas Electronics Corp. through various means, including the Renesas Electronics Corp. website (http://www.renesas.com).

Renesas Electronics
www.renesas.com

Rev.1.01 Sep 2015

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
3. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics product.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.

   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots etc.

   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; and safety equipment etc.

   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (nuclear reactor control systems, military equipment etc.). You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application for which it is not intended. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You should not use Renesas Electronics products or technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. When exporting the Renesas Electronics products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, who distributes, disposes of, or otherwise places the product with a third party, to notify such third party in advance of the contents and conditions set forth in this document, Renesas Electronics assumes no responsibility for any losses incurred by you or third parties as a result of unauthorized use of Renesas Electronics products.
11. This document may not be reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(2012.4)

How to Use This Manual

Readers

This manual is intended for users who design and develop application systems using RH850 family products.

Purpose

This manual is intended for users to understand the functions of real-time OS "RI850V4" manufactured by Renesas Electronics, described the organization listed below.

Organization

This manual can be broadly divided into the following units.

CHAPTER 1 OVERVIEW
CHAPTER 2 SYSTEM CONSTRUCTION
CHAPTER 3 TASK MANAGEMENT FUNCTIONS
CHAPTER 4 TASK DEPENDENT SYNCHRONIZATION FUNCTIONS
CHAPTER 5 SYNCHRONIZATION AND COMMUNICATION FUNCTIONS
CHAPTER 6 EXTENDED SYNCHRONIZATION AND COMMUNICATION FUNCTIONS
CHAPTER 7 MEMORY POOL MANAGEMENT FUNCTIONS
CHAPTER 8 SYSTEM STATE MANAGEMENT FUNCTIONS
CHAPTER 9 TIME MANAGEMENT FUNCTIONS
CHAPTER 10 INTERRUPT MANAGEMENT FUNCTIONS
CHAPTER 11 SERVICE CALL MANAGEMENT FUNCTIONS
CHAPTER 12 SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS
CHAPTER 13 SCHEDULER
CHAPTER 14 SYSTEM INITIALIZATION ROUTINE
CHAPTER 15 DATA TYPES AND MACROS
CHAPTER 16 SERVICE CALLS
CHAPTER 17 SYSTEM CONFIGURATION FILE
CHAPTER 18 CONFIGURATOR CF850V4
APPENDIX A WINDOW REFERENCE
APPENDIX B SIZE OF MEMORY
APPENDIX C SUPPORT FOR FLOATING-POINT OPERATION COPROCESSOR

How to Read This Manual

It is assumed that the readers of this manual have general knowledge in the fields of electrical engineering, logic circuits, microcontrollers, C language, and assemblers.

To understand the hardware functions of the RH850 family.
-> Refer to the **User's Manual** of each product.

Conventions

Data significance:	Higher digits on the left and lower digits on the right
Note:	Footnote for item marked with **Note** in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numeric representation:	Decimal ... XXXX
	Hexadecimal ... 0xXXXX

Prefixes indicating power of 2 (address space and memory capacity):

K (kilo) $2^{10} = 1024$

M (mega) $2^{20} = 1024^2$

Related Documents

The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

Document Name		Document No.
RI Series	Start	R20UT0751E
	Message	R20UT0756E
RI850V4 V2	Coding	This manual
	Debug	R20UT2890E
	Analysis	R20UT2891E

Caution The related documents listed above are subject to change without notice. Be sure to use the latest edition of each document when designing.

All trademarks or registered trademarks in this document are the property of their respective owners.

TABLE OF CONTENTS

CHAPTER 1 OVERVIEW ... 11

1.1 Outline ... 11
1.1.1 Real-Time OS ... 11
1.1.2 Multi-task OS ... 11
1.1.3 Support for RH850 multi-core devices ... 11
1.2 Execution Environment ... 12

CHAPTER 2 SYSTEM CONSTRUCTION ... 13

2.1 Outline ... 13
2.2 Cording System Configuration File ... 14
2.3 Coding Processing Programs ... 14
2.4 Coding User-Own Coding Module ... 15
2.5 Trace Information File ... 15
2.6 Creating Load Module ... 16
2.7 Option Settings for Build ... 20

CHAPTER 3 TASK MANAGEMENT FUNCTIONS ... 21

3.1 Outline ... 21
3.2 Tasks ... 21
3.2.1 Task state ... 21
3.2.2 Task priority ... 23
3.2.3 Basic form of tasks ... 24
3.2.4 Internal processing of task ... 25
3.3 Create Task ... 26
3.4 Activate Task ... 26
3.4.1 Queuing an activation request ... 26
3.4.2 Not queuing an activation request ... 27
3.5 Cancel Task Activation Requests ... 28
3.6 Terminate Task ... 29
3.6.1 Terminate invoking task ... 29
3.6.2 Terminate task ... 30
3.7 Change Task Priority ... 31
3.8 Reference Task Priority ... 32
3.9 Reference Task State ... 33
3.9.1 Reference task state ... 33
3.9.2 Reference task state (simplified version) ... 34
3.10 Memory Saving ... 35
3.10.1 Disable preempt ... 35

CHAPTER 4 TASK DEPENDENT SYNCHRONIZATION FUNCTIONS ... 36

4.1 Outline ... 36
4.2 Put Task to Sleep ... 36
4.2.1 Waiting forever ... 36
4.2.2 With timeout ... 38
4.3 Wakeup Task ... 39
4.4 Cancel Task Wakeup Requests ... 40
4.5 Release Task from Waiting ... 41
4.6 Suspend Task ... 42
4.7 Resume Suspended Task ... 43
4.7.1 Resume suspended task ... 43
4.7.2 Forcibly resume suspended task ... 44
4.8 Delay Task ... 45
4.9 Differences Between Wakeup Wait with Timeout and Time Elapse Wait ... 46

CHAPTER 5 SYNCHRONIZATION AND COMMUNICATION FUNCTIONS ... 47

5.1 Outline ... 47
5.2 Semaphores ... 47
5.2.1 Create semaphore ... 47
5.2.2 Acquire semaphore resource ... 48
5.2.3 Release semaphore resource ... 51
5.2.4 Reference semaphore state ... 52
5.3 Eventflags ... 53
5.3.1 Create eventflag ... 53
5.3.2 Set eventflag ... 54
5.3.3 Clear eventflag ... 55
5.3.4 Wait for eventflag ... 56
5.3.5 Reference eventflag state ... 61
5.4 Data Queues ... 62
5.4.1 Create data queue ... 62
5.4.2 Send to data queue ... 63
5.4.3 Forced send to data queue ... 68
5.4.4 Receive from data queue ... 69
5.4.5 Reference data queue state ... 74
5.5 Mailboxes ... 75
5.5.1 Messages ... 75
5.5.2 Create mailbox ... 76
5.5.3 Send to mailbox ... 77
5.5.4 Receive from mailbox ... 78
5.5.5 Reference mailbox state ... 81

CHAPTER 6 EXTENDED SYNCHRONIZATION AND COMMUNICATION FUNCTIONS ... 82

6.1 Outline ... 82
6.2 Mutexes ... 82
6.2.1 Differences from semaphores ... 82
6.2.2 Create mutex ... 83
6.2.3 Lock mutex ... 84
6.2.4 Unlock mutex ... 87
6.2.5 Reference mutex state ... 88

CHAPTER 7 MEMORY POOL MANAGEMENT FUNCTIONS ... 89

7.1 Outline ... 89
7.2 User-Own Coding Module ... 89
7.2.1 Post-overflow processing ... 90
7.3 Fixed-Sized Memory Pools ... 91
7.3.1 Create fixed-sized memory pool ... 91
7.3.2 Acquire fixed-sized memory block ... 92
7.3.3 Release fixed-sized memory block ... 97
7.3.4 Reference fixed-sized memory pool state ... 98
7.4 Variable-Sized Memory Pools ... 99
7.4.1 Create variable-sized memory pool ... 99
7.4.2 Acquire variable-sized memory block ... 100
7.4.3 Release variable-sized memory block ... 105
7.4.4 Reference variable-sized memory pool state ... 106

CHAPTER 8 SYSTEM STATE MANAGEMENT FUNCTIONS ... 107

8.1 Outline ... 107
8.2 Rotate Task Precedence ... 107
8.3 Forced Scheduler Activation ... 109
8.4 Reference Task ID in the RUNNING State ... 110
8.5 Lock the CPU ... 111
8.6 Unlock the CPU ... 113
8.7 Reference CPU State ... 115
8.8 Disable Dispatching ... 116
8.9 Enable Dispatching ... 118
8.10 Reference Dispatching State ... 120
8.11 Reference Contexts ... 121
8.12 Reference Dispatch Pending State ... 122

CHAPTER 9 TIME MANAGEMENT FUNCTIONS ... 123

9.1 Outline ... 123
9.2 System Time ... 123
9.2.1 Base clock timer interrupt ... 123
9.2.2 Base clock interval ... 124
9.3 Timer Operations ... 124
9.3.1 Delayed task wakeup ... 124
9.3.2 Timeout ... 124
9.3.3 Cyclic handlers ... 124
9.3.4 Create cyclic handler ... 125
9.4 Set System Time ... 126
9.5 Reference System Time ... 127
9.6 Start Cyclic Handler Operation ... 128
9.7 Stop Cyclic Handler Operation ... 130
9.8 Reference Cyclic Handler State ... 131

CHAPTER 10 INTERRUPT MANAGEMENT FUNCTIONS ... 132

10.1 Outline ... 132
10.2 User-Own Coding Module ... 132
10.2.1 Interrupt entry processing ... 132
10.3 Interrupt Handlers ... 134
10.3.1 Basic form of interrupt handlers ... 134
10.3.2 Internal processing of interrupt handler ... 134
10.3.3 Define interrupt handler ... 135
10.4 Base Clock Timer Interrupts ... 135
10.5 Multiple Interrupts ... 135

CHAPTER 11 SERVICE CALL MANAGEMENT FUNCTIONS ... 137

11.1 Outline ... 137
11.2 Extended Service Call Routines ... 137
11.2.1 Basic form extended service call routines ... 137
11.2.2 Internal processing of extended service call routine ... 138
11.3 Define Extended Service Call Routine ... 138
11.4 Invoke Extended Service Call Routine ... 139

CHAPTER 12 SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS ... 140

12.1 Outline ... 140
12.2 User-Own Coding Module ... 140
12.2.1 Initialization routine ... 140
12.2.2 Define initialization routine ... 141

CHAPTER 13 SCHEDULER ... 142

13.1 Outline ... 142
13.1.1 Drive Method ... 142
13.1.2 Scheduling Method ... 142
13.1.3 Ready queue ... 143
13.1.4 Scheduling Lock Function ... 143
13.2 User-Own Coding Module ... 145
13.2.1 Idle Routine ... 145
13.2.2 Define Idle Routine ... 146
13.3 Scheduling in Non-Tasks ... 146

CHAPTER 14 SYSTEM INITIALIZATION ROUTINE ... 147

14.1 Outline ... 147
14.2 User-Own Coding Module ... 148
14.2.1 Boot processing ... 148
14.2.2 System dependent information ... 150
14.3 Kernel Initialization Module ... 152

CHAPTER 15 DATA TYPES AND MACROS ... 153

15.1 Data Types ... 153
15.2 Packet Formats ... 155

15.2.1 Task state packet ... 155
15.2.2 Task state packet (simplified version) ... 157
15.2.3 Semaphore state packet ... 158
15.2.4 Eventflag state packet ... 159
15.2.5 Data queue state packet ... 160
15.2.6 Message packet ... 161
15.2.7 Mailbox state packet ... 162
15.2.8 Mutex state packet ... 163
15.2.9 Fixed-sized memory pool state packet ... 164
15.2.10 Variable-sized memory pool state packet ... 165
15.2.11 System time packet ... 166
15.2.12 Cyclic handler state packet ... 167
15.3 Data Macros ... 168
15.3.1 Current state ... 168
15.3.2 Processing program attributes ... 169
15.3.3 Management object attributes ... 169
15.3.4 Service call operating modes ... 170
15.3.5 Return value ... 170
15.3.6 Kernel configuration constants ... 171
15.4 Conditional Compile Macro ... 172

CHAPTER 16 SERVICE CALLS ... 173

16.1 Outline ... 173
16.1.1 Call service call ... 174
16.2 Explanation of Service Call ... 175
16.2.1 Task management functions ... 178
16.2.2 Task dependent synchronization functions ... 192
16.2.3 Synchronization and communication functions (semaphores) ... 205
16.2.4 Synchronization and communication functions (eventflags) ... 214
16.2.5 Synchronization and communication functions (data queues) ... 225
16.2.6 Synchronization and communication functions (mailboxes) ... 239
16.2.7 Extended synchronization and communication functions (mutexes) ... 249
16.2.8 Memory pool management functions (fixed-sized memory pools) ... 258
16.2.9 Memory pool management functions (variable-sized memory pools) ... 269
16.2.10 Time management functions ... 280
16.2.11 System state management functions ... 288
16.2.12 Service call management functions ... 301

CHAPTER 17 SYSTEM CONFIGURATION FILE ... 303

17.1 Outline ... 303
17.2 Configuration Information ... 305
17.2.1 Cautions ... 306
17.3 Declarative Information ... 307
17.3.1 Header file declaration ... 307
17.4 System Information ... 308
17.4.1 RI series information ... 308
17.4.2 Basic information ... 309

17.4.3 FPSR register information ... 311
17.4.4 Memory area information ... 312
17.5 Static API Information ... 313
17.5.1 Task information ... 313
17.5.2 Semaphore information ... 315
17.5.3 Eventflag information ... 316
17.5.4 Data queue information ... 317
17.5.5 Mailbox information ... 318
17.5.6 Mutex information ... 319
17.5.7 Fixed-sized memory pool information ... 320
17.5.8 Variable-sized memory pool information ... 321
17.5.9 Cyclic handler information ... 322
17.5.10 Interrupt handler information ... 324
17.5.11 Extended service call routine information ... 325
17.5.12 Initialization routine information ... 326
17.5.13 Idle routine information ... 327
17.6 Description Examples ... 328

CHAPTER 18 CONFIGURATOR CF850V4 ... 329

18.1 Outline ... 329
18.2 Activation Method ... 330
18.2.1 Activating from command line ... 330
18.2.2 Activating from CS+ ... 333
18.2.3 Command file ... 334
18.2.4 Command input examples ... 335

APPENDIX A WINDOW REFERENCE ... 336

A.1 Description ... 336

APPENDIX B SIZE OF MEMORY ... 353

B.1 Description ... 353
B.1.1 .kernel_system ... 353
B.1.2 .kernel_const ... 355
B.1.3 .kernel_data ... 356
B.1.4 .kernel_data_init ... 357
B.1.5 .kernel_const_trace.const ... 357
B.1.6 .kernel_data_trace.bss ... 358
B.1.7 .kernel_work ... 359
B.1.8 .sec_nam(user-definied area) ... 361

APPENDIX C SUPPORT FOR FLOATING-POINT OPERATION COPROCESSOR ... 362

CHAPTER 1 OVERVIEW

1.1 Outline

The RI850V4 is a built-in real-time, multi-task OS that provides a highly efficient real-time, multi-task environment to increases the application range of processor control units.

The RI850V4 is a high-speed, compact OS capable of being stored in and run from the ROM of a target system.

It can also be used in RH850 multi-core devices.

1.1.1 Real-Time OS

Control equipment demands systems that can rapidly respond to events occurring both internal and external to the equipment. Conventional systems have utilized simple interrupt handling as a means of satisfying this demand. As control equipment has become more powerful, however, it has proved difficult for systems to satisfy these requirements by means of simple interrupt handling alone.

In other words, the task of managing the order in which internal and external events are processed has become increasingly difficult as systems have increased in complexity and programs have become larger.

Real-time OS has been designed to overcome this problem.

The main purpose of a real-time OS is to respond to internal and external events rapidly and execute programs in the optimum order.

1.1.2 Multi-task OS

A "task" is the minimum unit in which a program can be executed by an OS. "Multi-task" is the name given to the mode of operation in which a single processor processes multiple tasks concurrently.

Actually, the processor can handle no more than one program (instruction) at a time. But, by switching the processor’s attention to individual tasks on a regular basis (at a certain timing) it appears that the tasks are being processed simultaneously.

A multi-task OS enables the parallel processing of tasks by switching the tasks to be executed as determined by the system.

One important purpose of a multi-task OS is to improve the throughput of the overall system through the parallel processing of multiple tasks.

1.1.3 Support for RH850 multi-core devices

The RI850V4 supports build processing for multi-core devices. The target processor element (PE) where the RI850V4 is to be used can be specified and the RI850V4 can be used in multiple PEs at the same time.

The RI850V4 is a real-time OS for a single core, which is intended to operate in a single PE, and it does not provide facilities for controlling the processing between PEs.

As a measure for implementing the control of the processing between PEs, a library specialized for multi-core devices can be used. Renesas Electronics offers the "libipcx library for communication and exclusive control between processor elements" (hereafter called libipcx), which is a sample library supporting the RH850 multi-core devices. Using the RI850V4 and libipcx together enables control of the processing between PEs.

1.2 Execution Environment

The RI850V4 supports the RH850 family (G3K core and G3M core).

The following is a list of reserved OS resources that are exclusively used by the RI850V4 and cannot be modified from processing programs.

Reserved OS Resources
General register (r2)
OS timer (OSTM): one channel
Interrupt priority mask (PMR)
UM bit in the program status word (PSW)
Interrupt configurations (INTCFG)
Exception handler vector address (EBASE)
Base address of the interrupt handler table (INTBP)

Note Whether the exception handler vector address (EBASE) or the base address of the interrupt handler table (INTBP) is reserved depends on the option settings for activation of the CONFIGURATOR CF850V4. When -ebase= <*Exception Base Address*> is specified, the exception handler vector address (EBASE) is handled as a reserved resource; when -intbp=<*Interrupt Base Address*> is specified, the base address of the interrupt handler table (INTBP) is handled as a reserved resource.

CHAPTER 2 SYSTEM CONSTRUCTION

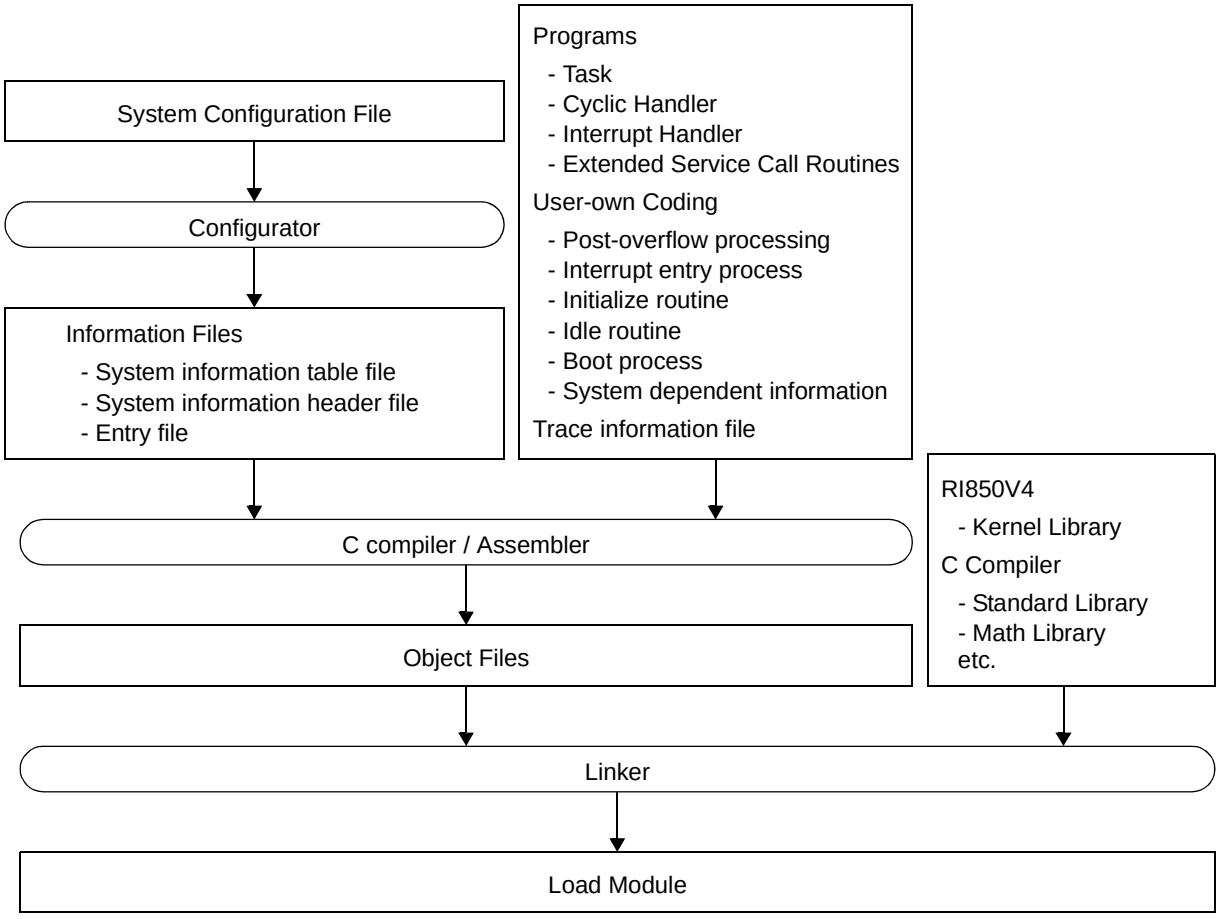
This chapter describes how to build a system (load module) that uses the functions provided by the RI850V4.

2.1 Outline

System building consists in the creation of a load module using the files (kernel library, etc.) installed on the user development environment (host machine) from the RI850V4's supply media.

The following shows the procedure for organizing the system.

Figure 2-1 Example of System Construction



The RI850V4 provides a sample program with the files necessary for generating a load module.

For the location where the sample program is stored, see "RI Series Real-Time Operating System User's Manual: Start".

2.2 Cording System Configuration File

Code the SYSTEM CONFIGURATION FILE required for creating information files (system information table file, system information header file, entry file) that contain data to be provided for the RI850V4.

Note For details about the system configuration file, refer to "CHAPTER 17 SYSTEM CONFIGURATION FILE".

2.3 Coding Processing Programs

Code the processing that should be implemented in the system.
In the RI850V4, the processing program is classified into the following seven types, in accordance with the types and purposes of the processing that should be implemented.

- Tasks
  A task is processing program that is not executed unless it is explicitly manipulated via service calls provided by the RI850V4, unlike other processing programs (cyclic handler, interrupt handler, etc.).

- Cyclic handlers
  The cyclic handler is a routine dedicated to cycle processing that is activated periodically at a constant interval (activation cycle).
  The RI850V4 handles the cyclic handler as a "non-task (module independent from tasks)".
  Therefore, even if a task with the highest priority in the system is being executed, the processing is suspended when a specified activation cycle has come, and the control is passed to the cyclic handler.

- Interrupt Handlers
  The interrupt handler is a routine dedicated to interrupt servicing that is activated when an EI level maskable interrupt occurs.
  The RI850V4 handles the interrupt handler as a "non-task (module independent from tasks)".
  Therefore, even if a task with the highest priority in the system is being executed, its processing is suspended when an EI level maskable interrupt occurs, and control is passed to the interrupt handler.

- Extended Service Call Routines
  This is a routine to which user-defined functions are registered in the RI850V4, and will never be executed unless it is called explicitly, using service calls provided by the RI850V4.
  The RI850V4 positions extended service call routines as extensions of the processing program that called the extended service call routine.

Note For details about the processing programs, refer to "CHAPTER 3 TASK MANAGEMENT FUNCTIONS", "CHAPTER 9 TIME MANAGEMENT FUNCTIONS", "CHAPTER 10 INTERRUPT MANAGEMENT FUNCTIONS", "CHAPTER 11 SERVICE CALL MANAGEMENT FUNCTIONS".

2.4 Coding User-Own Coding Module

To support various execution environments, the hardware-dependent processing and various information required for the RI850V4 to execute processing are extracted as user-own coding modules.

This enhances portability for various execution environments and facilitates customization as well.

The user-own coding modules for the RI850V4 are classified into the following six types depending on the type of hardware-dependent processing to be executed and the usage of the module.

- Post-overflow processing
  A routine dedicated to post-processing (function name: _kernel_stk_overflow) that is extracted as a user-own coding module to execute post-overflow processing and is called when a stack overflow occurs in the RI850V4 or a processing program.
  Acceptance of interrupts is disabled (the ID flag in the program status word (PSW) is set to 1) in the initial state after activation.

- Interrupt entry processing
  A routine dedicated to entry processing that is extracted as a user-own coding module to assign processing for branching to the relevant processing (such as interrupt preprocessing), to the handler address to which the CPU forcibly passes control when an interrupt occurs.
  The interrupt entry processing for the EI level maskable interrupts defined in the Interrupt handler information in the system configuration file is included in the entry file created by executing the configurator for the system configuration file.
  Therefore, coding of interrupt entry processing is necessary for other interrupts (such as a reset) that are not EI level maskable interrupts.

- Initialization routine
  A routine dedicated to initialization processing that is extracted as a user-own coding module to initialize the hardware dependent on the user execution environment (such as the peripheral controller), and is called from the Kernel Initialization Module.

- Idle Routine
  A routine dedicated to idle processing that is extracted from the SCHEDULER as a user-own coding module to utilize the standby function provided by the CPU (to achieve the low-power consumption system), and is called from the scheduler when there no longer remains a task subject to scheduling by the RI850V4 (task in the RUNNING or READY state) in the system.

- Boot processing
  A routine dedicated to initialization processing that is extracted as a user-own coding module to initialize the minimum required hardware for the RI850V4 to perform processing, and is called from Interrupt entry processing.

- System dependent information
  The system-dependent information is a header file (file name: userown.h) including various information required for the RI850V4 to execute processing, which is extracted as a user-own coding module.

Note For details about the user-own coding module, refer to “CHAPTER 7 MEMORY POOL MANAGEMENT FUNCTIONS“, "CHAPTER 10 INTERRUPT MANAGEMENT FUNCTIONS", "CHAPTER 12 SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS", "CHAPTER 13 SCHEDULER", "CHAPTER 14 SYSTEM INITIALIZATION ROUTINE".

2.5 Trace Information File

The trace information file (file name: trcinf.c) includes descriptions of the processing necessary for the trace mode selected in the Property panel -> [Task Analyzer] tabbed page -> [Trace] category -> [Selection of trace mode].

The user does not need to modify the contents of this file.

Note that this file should be incorporated into the load module even when the trace facility is not used. Include this file as a target of build processing even when using the GHS-version development environment.

2.6 Creating Load Module

Run a build on CS+ for files created in sections from "2.2 Cording System Configuration File" to "2.4 Coding User-Own Coding Module", the trace information file, and the library files provided by the RI850V4 and C compiler package, to create a load module.

1) Create or load a project

Create a new project, or load an existing one.

Note See RI Series Real-time OS User's Manual: Start or CS+ Integrated Development Environment User's Manual: Start for details about creating a new project or loading an existing one.

2) Set a build target project

Specify a project as the target of build.

Note See CS+ Integrated Development Environment User's Manual: RH850 Build for details about setting the active project.

3) Set build target files

For the project, add or remove build target files and update the dependencies.

Note See CS+ Integrated Development Environment User's Manual: RH850 Build for details about adding or removing build target files for the project and updating the dependencies.

The following lists the files required for creating a load module.

- System configuration file created in "CHAPTER 2.2 Cording System Configuration File"
  - SYSTEM CONFIGURATION FILE

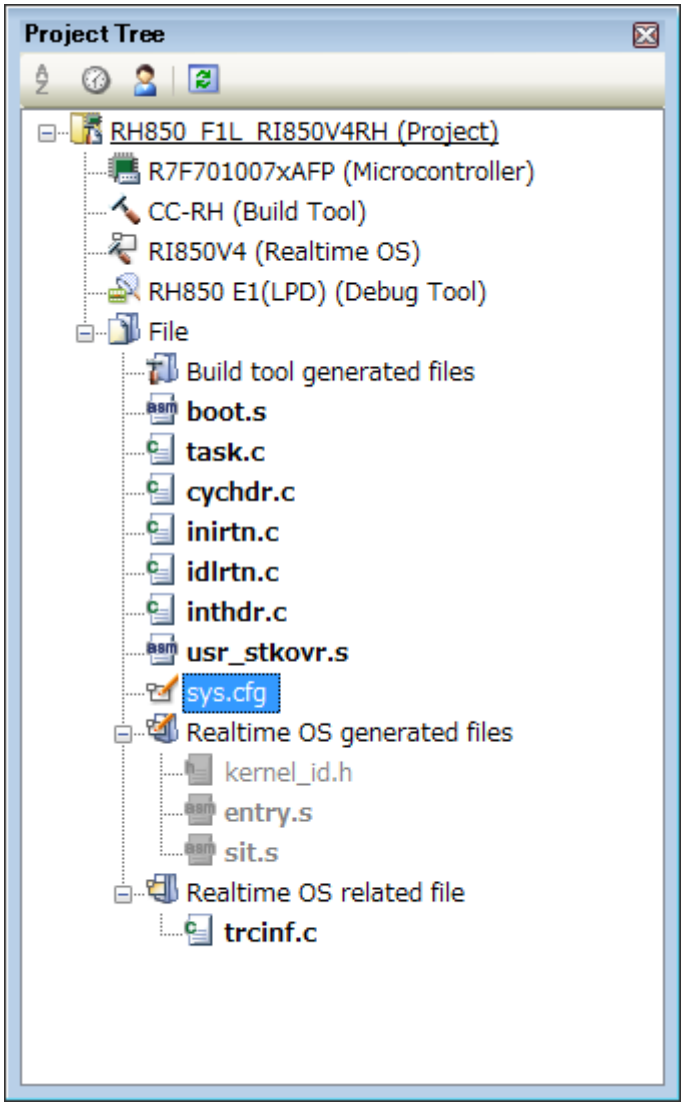
  Note Specify "cfg" as the extension of the system configuration file name. If the extension is different, "cfg" is automatically added (for example, if you designate "aaa.c" as a file name, the file is named as "aaa.c.cfg").

- C/assembly language source files created in "2.3 Coding Processing Programs"
  - Processing programs (tasks, cyclic handlers, interrupt handlers, extended service call routines)
- C/assembly language source files created in "2.4 Coding User-Own Coding Module"
  - User-own coding module (post-overflow processing, interrupt entry processing, initialization routine, idle routine, boot processing, system dependent information)
- Trace information files provided by the RI850V4
- Library files provided by the RI850V4
  - Kernel library
- Library files provided by the C compiler package
  - Standard library, Math library, etc.

Note 1 If the system configuration file is added to the Project Tree panel, the Real-Time OS generated files node is appeared.
The following information files are appeared under the Real-Time OS generated files node. However, these files are not generated at this point in time.

- System information table file
- System information header file
- Entry file

Figure 2-2 Project Tree Panel (After Adding sys.cfg)



Note 2 When replacing the system configuration file, first remove the added system configuration file from the project, then add another one again.

Note 3 Although it is possible to add more than one system configuration files to a project, only the first file added is enabled. Note that if you remove the enabled file from the project, the remaining additional files will not be enabled; you must therefore add them again.

4) Specify the output of a load module file

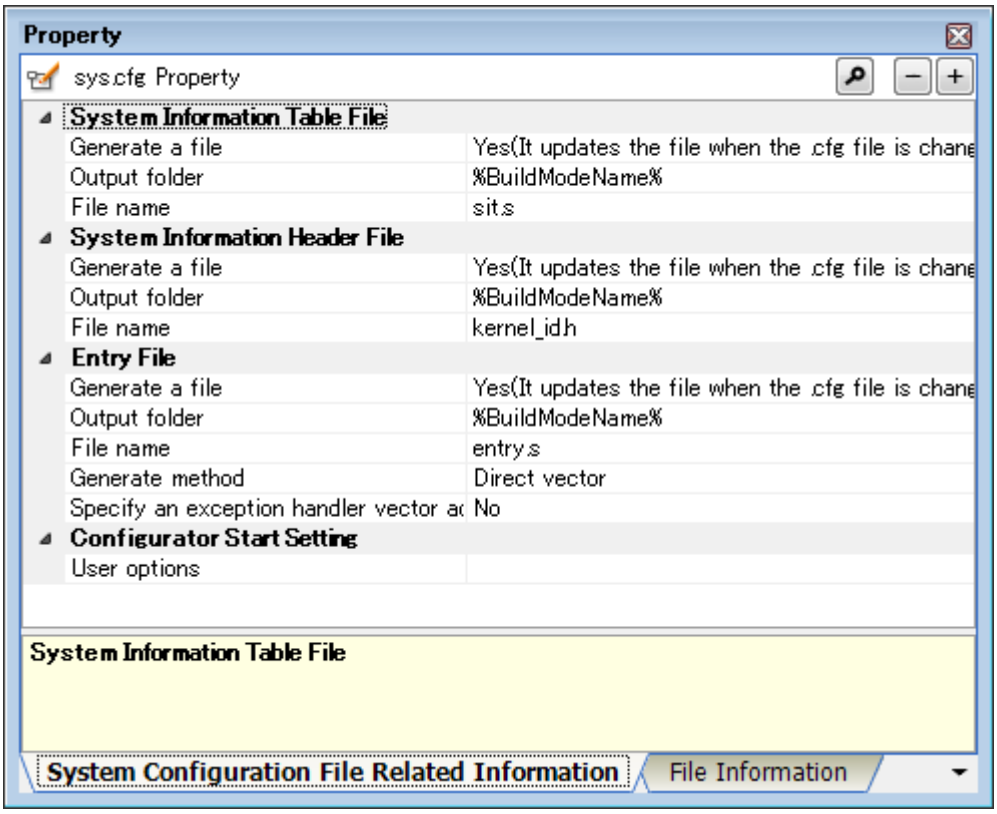
Specify the type of load module file to be generated.

Note For details of the load module output settings, see "CS+ Integrated Development Environment User's Manual: CC-RH Build Tool Operation".

5) Set the output of information files

Select the system configuration file on the project tree to open the Property panel.
On the [System Configuration File Related Information] tab, set the output of information files (system information table file, system information header file, and entry file).

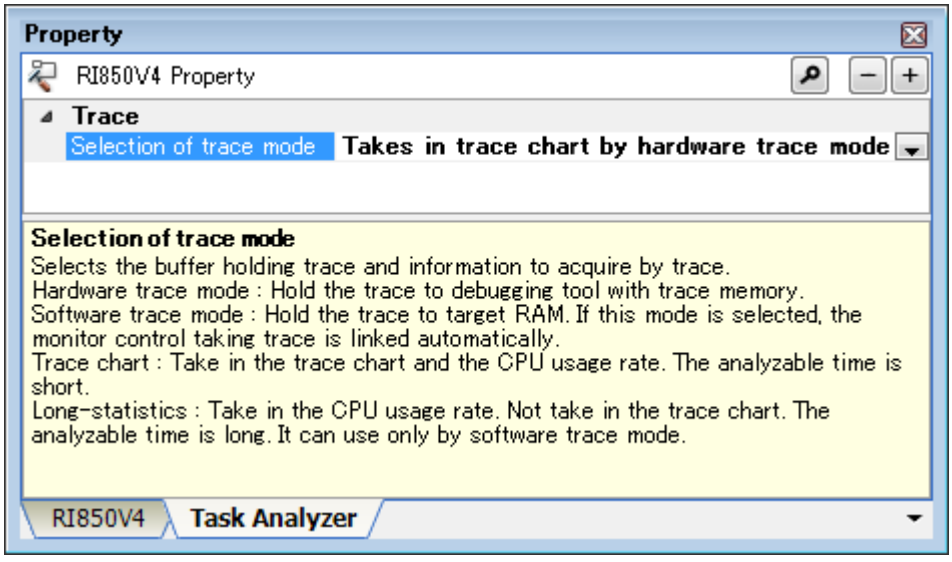
Figure 2-3 Property Panel: [System Configuration File Related Information] Tab



6) Set trace function

Use the task analyzer tool (a utility tool provided by the RI850V4) on the [Task Analyzer] tabbed page in the Property panel to specify the information necessary to analyze the execution history (trace data) of the processing program.

Figure 2-4 [Task Analyzer] Tab


Property
RI850V4 Property
Trace
Selection of trace mode
Takes in trace chart by hardware trace mode
Selection of trace mode
Selects the buffer holding trace and information to acquire by trace.
Hardware trace mode : Hold the trace to debugging tool with trace memory.
Software trace mode : Hold the trace to target RAM. If this mode is selected, the monitor control taking trace is linked automatically.
Trace chart : Take in the trace chart and the CPU usage rate. The analyzable time is short.
Long-statistics : Take in the CPU usage rate. Not take in the trace chart. The analyzable time is long. It can use only by software trace mode.
RI850V4
Task Analyzer

7) Set build options

Set the options for the compiler, assembler, linker, and the like.

When using the RI850V4, some options should always be specified. For details, see "15.4 Conditional Compile Macro".

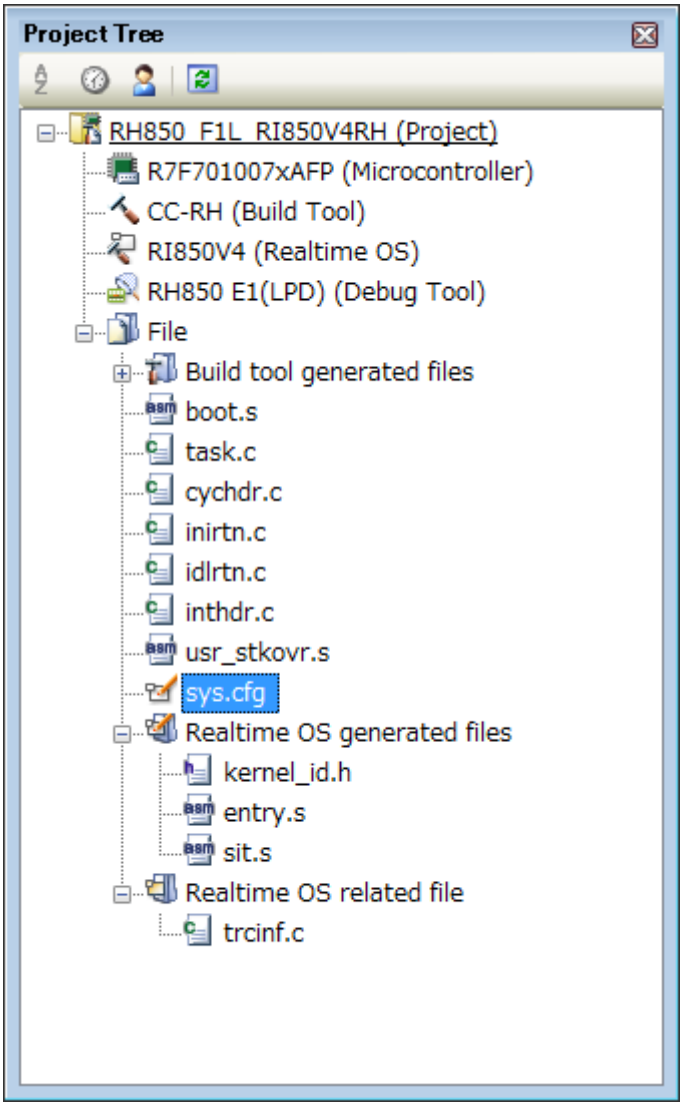
Note See CS+ Integrated Development Environment User's Manual: RH850 Build for details about setting build options.

8) Run a build

Run a build to create a load module.

Note See CS+ Integrated Development Environment User's Manual: RH850 Build for details about running a build.

Figure 2-5 Project Tree Panel (After Running Build)



9) Save the project

Save the setting information of the project to the project file.

Note See CS+ Integrated Development Environment User's Manual: Project Operation for details about saving the project.

2.7 Option Settings for Build

When using the RI850V4, the following options should always be specified for user applications.
In addition, the options listed in "15.4 Conditional Compile Macro" should also be specified when using the header file provided by the RI850V4.

- CC-RH version

Build Option	Description
-Xreserve_r2	Reserves the r2 register.
-D__rel__	Definition of the compiler from Renesas Electronics. Add two underscores before and after "rel".
-Xep=callee	Specifies the handling of the EP register.

- CCV850 version

Build Option	Description
-reserve_r2	Reserves the r2 register.
-D__ghs__	Definition of the compiler from Green Hills Software. Add two underscores before and after "ghs".

CHAPTER 3 TASK MANAGEMENT FUNCTIONS

This chapter describes the task management functions performed by the RI850V4.

3.1 Outline

The task management functions provided by the RI850V4 include a function to reference task statuses such as priorities and detailed task information, in addition to a function to manipulate task statuses such as generation, activation and termination of tasks.

3.2 Tasks

A task is processing program that is not executed unless it is explicitly manipulated via service calls provided by the RI850V4, unlike other processing programs (cyclic handler and interrupt handler), and is called from the scheduler.

The RI850V4 manages the states in which each task may enter and tasks themselves, by using management objects (task management blocks) corresponding to tasks one-to-one.

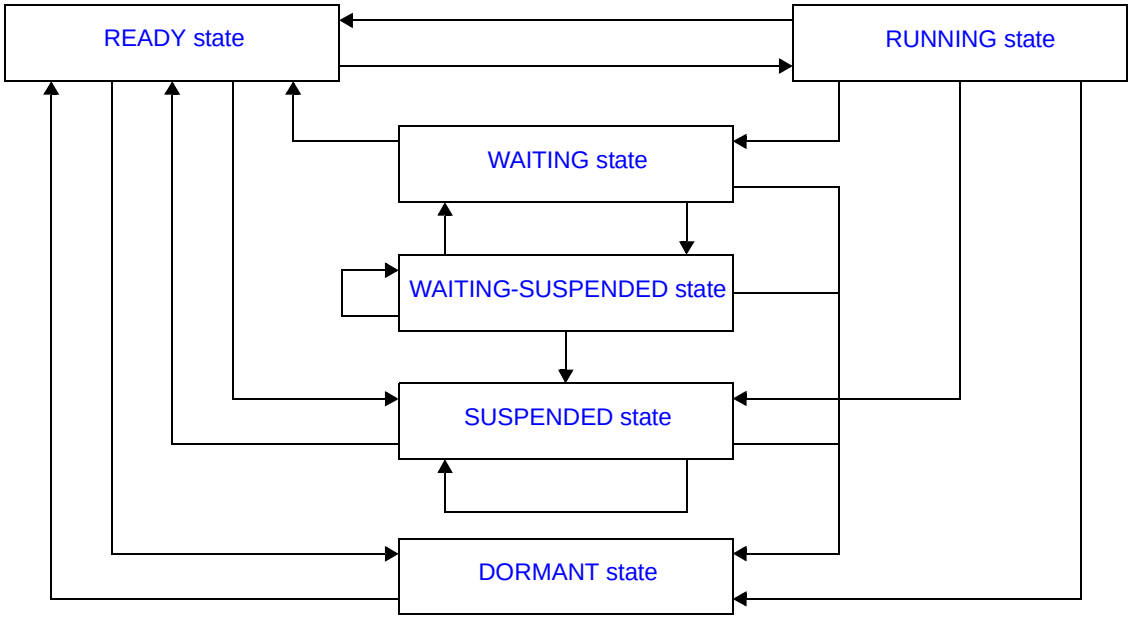
Note The execution environment information required for a task's execution is called "task context". During task execution switching, the task context of the task currently under execution by the RI850V4 is saved and the task context of the next task to be executed is loaded.

3.2.1 Task state

Tasks enter various states according to the acquisition status for the OS resources required for task execution and the occurrence/non-occurrence of various events. In this process, the current state of each task must be checked and managed by the RI850V4.

The RI850V4 classifies task states into the following six types.

Figure 3-1 Task State



1) DORMANT state

State of a task that is not active, or the state entered by a task when processing has ended.
A task in the DORMANT state, while being under management of the RI850V4, is not subject to RI850V4 scheduling.

2) READY state

State of a task for which the preparations required for processing execution have been completed, but since another task with a higher priority level or a task with the same priority level is currently being processed, the task is waiting to be given the CPU's use right.

3) RUNNING state

State of a task that has acquired the CPU use right and is currently being processed.
Only one task can be in the running state at one time in the entire system.

4) WAITING state

State in which processing execution has been suspended because conditions required for execution are not satisfied.
Resumption of processing from the WAITING state starts from the point where the processing execution was suspended. The value of information required for resumption (such as task context) immediately before suspension is therefore restored.
In the RI850V4, the WAITING state is classified into the following ten types according to their required conditions and managed.

Table 3-1 WAITING State

WAITING State	Description
Sleeping state	A task enters this state if the counter for the task (registering the number of times the wakeup request has been issued) indicates 0x0 upon the issue of slp_tsk or tslp_tsk.
Delayed state	A task enters this state upon the issue of a dly_tsk.
WAITING state for a semaphore resource	A task enters this state if it cannot acquire a resource from the relevant semaphore upon the issue of wai_sem or twai_sem.
WAITING state for an eventflag	A task enters this state if a relevant eventflag does not satisfy a predetermined condition upon the issue of wai_flg or twai_flg.
Sending WAITING state for a data queue	A task enters this state if cannot send a data to the relevant data queue upon the issue of snd_dtq or tsnd_dtq.
Receiving WAITING state for a data queue	A task enters this state if cannot receive a data from the relevant data queue upon the issue of rcv_dtq or trcv_dtq.
Receiving WAITING state for a mailbox	A task enters this state if cannot receive a message from the relevant mailbox upon the issue of rcv_mbx or trcv_mbx.
WAITING state for a mutex	A task enters this state if cannot lock the relevant mutex upon the issue of loc_mtx or tloc_mtx.
WAITING state for a fixed-sized memory block	A task enters this state if it cannot acquire a fixed-sized memory block from the relevant fixed-sized memory pool upon the issue of get_mpf or tget_mpf.
WAITING state for a variable-sized memory block	A task enters this state if it cannot acquire a variable-sized memory block from the relevant variable-sized memory pool upon the issue of get_mpl or tget_mpl.

5) SUSPENDED state

State in which processing execution has been suspended forcibly.
Resumption of processing from the SUSPENDED state starts from the point where the processing execution was suspended. The value of information required for resumption (such as task context) immediately before suspension is therefore restored.

6) WAITING-SUSPENDED state

State in which the WAITING and SUSPENDED states are combined.
A task enters the SUSPENDED state when the WAITING state is cancelled, or enters the WAITING state when the SUSPENDED state is cancelled.

3.2.2 Task priority

A priority level that determines the order in which that task will be processed in relation to the other tasks is assigned to each task.

As a result, in the RI850V4, the task that has the highest priority level of all the tasks that have entered an executable state (RUNNING state or READY state) is selected and given the CPU use right.

In the RI850V4, the following two types of priorities are used for management purposes.

- Initial priority
  Priority set when a task is created.
  Therefore, the priority level of a task (priority level referenced by the scheduler) immediately after it moves from the DORMANT state to the READY state is the initial priority.

- Current priority
  Priority referenced by the RI850V4 when it performs a manipulation (task scheduling, queuing tasks to a wait queue in the order of priority, or priority level inheritance) when a task is activated.

Note 1 In the RI850V4, a task having a smaller priority number is given a higher priority.

Note 2 The priority range that can be specified in a system can be defined in Basic information (Maximum priority: maxtpri) when creating a system configuration file.

3.2.3 Basic form of tasks

When coding a task, use a void function with one VP_INT argument (any function name is fine).

The extended information specified with Task information, or the start code specified when sta_tsk or ista_tsk is issued, is set for the *exinf* argument.

The following shows the basic form of tasks in C.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    .........
    .........

    ext_tsk ();                     /*Terminate invoking task*/
}
```

Note 1 If a task moves from the DORMANT state to the READY state by issuing sta_tsk or ista_tsk, the start code specified when issuing sta_tsk or ista_tsk is set to the *exinf* argument.

Note 2 When the return instruction is issued in a task, the same processing as ext_tsk is performed.

Note 3 For details about the extended information, refer to "3.4 Activate Task".

3.2.4 Internal processing of task

In the RI850V4, original dispatch processing (task scheduling) is executed during task switching.
Therefore, note the following points when coding tasks.

- Coding method
  Code tasks using C or assembly language.
  When coding in C, they can be coded in the same manner as ordinary functions coded.
  When coding in assembly language, code them according to the calling rules prescribed in the compiler used.

- Stack switching
  When switching tasks, the RI850V4 performs switching to the task specified in Task information.

- Service call issue
  Service calls that can be issued in tasks are limited to the service calls that can be issued from tasks.

  Note For details on the valid issue range of each service call, refer to Table 16-1 to Table 16-12.

- Acceptance of EI level maskable interrupts
  When a task is activated, the RI850V4 sets the interrupt acceptance status according to the settings in the Attribute: tskatr (such as the description language and initial state after activation) by manipulating the PM*n* bits in the priority mask register (PMR) and the ID bit in the program status word (PSW).

3.3 Create Task

In the RI850V4, the method of creating a task is limited to "static creation".

Tasks therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static task creation means defining of tasks using static API "CRE_TSK" in the system configuration file.

For details about the static API "CRE_TSK", refer to "17.5.1 Task information".

3.4 Activate Task

The RI850V4 provides two types of interfaces for task activation: queuing an activation request queuing and not queuing an activation request.

In the RI850V4, extended information specified in Task information during configuration and the value specified for the second parameter stacd when service call sta_tsk or ista_tsk is issued are called "extended information".

3.4.1 Queuing an activation request

A task (queuing an activation request) is activated by issuing the following service call from the processing program.

- act_tsk, iact_tsk

  These service calls move a task specified by parameter *tskid* from the DORMANT state to the READY state.

  As a result, the target task is queued at the end on the ready queue corresponding to the initial priority and becomes subject to scheduling by the RI850V4.

  If the target task has been moved to a state other than the DORMANT state when this service call is issued, this service call does not move the state but increments the activation request counter (by added 0x1 to the wakeup request counter).

  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System infromation header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = 8;          /*Declares and initializes variable*/

    .........
    .........

    act_tsk (tskid);            /*Activate task (queues an activation request)*/

    .........
    .........
}
```

Note 1 The activation request counter managed by the RI850V4 is configured in 7-bit widths. If the number of activation requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

Note 2 Extended information specified in Task information is passed to the task activated by issuing these service calls.

3.4.2 Not queuing an activation request

A task (not queuing an activation request) is activated by issuing the following service call from the processing program.

- sta_tsk, ista_tsk
  These service calls move a task specified by parameter *tskid* from the DORMANT state to the READY state.
  As a result, the target task is queued at the end on the ready queue corresponding to the initial priority and becomes subject to scheduling by the RI850V4.
  This service call does not perform queuing of activation requests. If the target task is in a state other than the DORMANT state, the status manipulation processing for the target task is therefore not performed but "E_OBJ" is returned.
  Specify for parameter *stacd* the extended information transferred to the target task.
  The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = 8;          /*Declares and initializes variable*/
    VP_INT  stacd = 123;        /*Declares and initializes variable*/

    .........
    .........

    sta_tsk (tskid, stacd);     /*Activate task (does not queue an activation */
                                /*request)*/


    .........
    .........
}
```

3.5 Cancel Task Activation Requests

An activation request is cancelled by issuing the following service call from the processing program.

- can_act, ican_act
  This service call cancels all of the activation requests queued to the task specified by parameter *tskid* (sets the activation request counter to 0x0).
  When this service call is terminated normally, the number of cancelled activation requests is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER_UINT ercd;                   /*Declares variable*/
    ID      tskid = 8;              /*Declares and initializes variable*/

    .........
    .........

    ercd = can_act (tskid);         /*Cancel task activation requests*/

    if (ercd >= 0x0) {
           .........                /*Normal termination processing*/
           .........
    }

    .........
    .........
}
```

Note This service call does not perform status manipulation processing but performs the setting of activation request counter. Therefore, the task does not move from a state such as the READY state to the DORMANT state.

3.6 Terminate Task

3.6.1 Terminate invoking task

An invoking task is terminated by issuing the following service call from the processing program.

- ext_tsk
  This service call moves an invoking task from the RUNNING state to the DORMANT state.
  As a result, the invoking task is unlinked from the ready queue and excluded from the RI850V4 scheduling subject.
  If an activation request has been queued to the invoking task (the activation request counter is not set to 0x0) when this service call is issued, this service call moves the task from the RUNNING state to the DORMANT state, decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter), and then moves the task from the DORMANT state to the READY state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    .........
    .........

    ext_tsk ();                     /*Terminate invoking task*/
}
```

Note 1 When moving a task from the RUNNING state to the DORMANT state, this service call initializes the following information to values that are set during task creation.

- Priority (current priority)
- Wakeup request count
- Suspension count
- Interrupt status

If an invoking task has locked a mutex, the locked state is released at the same time (processing equivalent to unl_mtx).

Note 2 When the return instruction is issued in a task, the same processing as ext_tsk is performed.

3.6.2 Terminate task

Other tasks are forcibly terminated by issuing the following service call from the processing program.

- ter_tsk
  This service call forcibly moves a task specified by parameter *tskid* to the DORMANT state.
  As a result, the target task is excluded from the RI850V4 scheduling subject.
  If an activation request has been queued to the target task (the activation request counter is not set to 0x0) when this service call is issued, this service call moves the task to the DORMANT state, decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter), and then moves the task from the DORMANT state to the READY state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/

    .........
    .........

    ter_tsk (tskid);                /*Terminate task*/

    .........
    .........
}
```

Note When moving a task to the DORMANT state, this service call initializes the following information to values that are set during task creation.

- Priority (current priority)
- Wakeup request count
- Suspension count
- Interrupt status

If the target task has locked a mutex, the locked state is released at the same time (processing equivalent to unl_mtx).

3.7 Change Task Priority

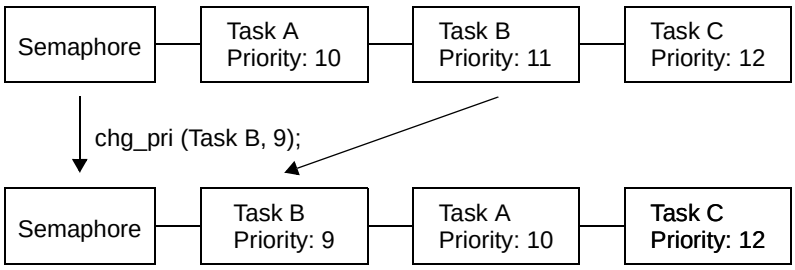
The priority is changed by issuing the following service call from the processing program.

- chg_pri, ichg_pri
  These service calls change the priority of the task specified by parameter *tskid* (current priority) to a value specified by parameter *tskpri*.
  If the target task is in the RUNNING or READY state after this service call is issued, this service call re-queues the task at the end of the ready queue corresponding to the priority specified by parameter *tskpri*, following priority change processing.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/
    PRI     tskpri = 9;             /*Declares and initializes variable*/

    .........
    .........

    chg_pri (tskid, tskpri);        /*Change task priority*/

    .........
    .........
}
```

Note When the target task is queued to a wait queue in the order of priority, the wait order may change due to issue of this service call.

Example When three tasks (task A: priority level 10, task B: priority level 11, task C: priority level 12) are queued to the semaphore wait queue in the order of priority, and the priority level of task B is changed from 11 to 9, the wait order will be changed as follows.



3.8 Reference Task Priority

A task priority is referenced by issuing the following service call from the processing program.

- get_pri, iget_pri
  Stores current priority of the task specified by parameter *tskid* in the area specified by parameter *p_tskpri*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/
    PRI     p_tskpri;               /*Declares variable*/

    .........
    .........

    get_pri (tskid, &p_tskpri);     /*Reference task priority*/

    .........
    .........
}
```

3.9 Reference Task State

3.9.1 Reference task state

A task status is referenced by issuing the following service call from the processing program.

- ref_tsk, iref_tsk
  Stores task state packet (current state, current priority, etc.) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtsk*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/
    T_RTSK  pk_rtsk;                /*Declares data structure*/
    STAT    tskstat;                /*Declares variable*/
    PRI     tskpri;                 /*Declares variable*/
    STAT    tskwait;                /*Declares variable*/
    ID      wobjid;                 /*Declares variable*/
    TMO     lefttmo;                /*Declares variable*/
    UINT    actcnt;                 /*Declares variable*/
    UINT    wupcnt;                 /*Declares variable*/
    UINT    suscnt;                 /*Declares variable*/
    ATR     tskatr;                 /*Declares variable*/
    PRI     itskpri;                /*Declares variable*/

    .........
    .........

    ref_tsk (tskid, &pk_rtsk);      /*Reference task state*/

    tskstat = pk_rtsk.tskstat;      /*Reference current state*/
    tskpri = pk_rtsk.tskpri;        /*Reference current priority*/
    tskwait = pk_rtsk.tskwait;      /*Reference reason for waiting*/
    wobjid = pk_rtsk.wobjid;        /*Reference object ID number for which the */
                                    /*task is waiting*/
    lefttmo = pk_rtsk.lefttmo;      /*Reference remaining time until timeout*/
    actcnt = pk_rtsk.actcnt;        /*Reference activation request count*/
    wupcnt = pk_rtsk.wupcnt;        /*Reference wakeup request count*/
    suscnt = pk_rtsk.suscnt;        /*Reference suspension count*/
    tskatr = pk_rtsk.tskatr;        /*Reference attribute*/
    itskpri = pk_rtsk.itskpri;      /*Reference initial priority*/

    .........
    .........
}
```

Note For details about the task state packet, refer to "15.2.1 Task state packet".

3.9.2 Reference task state (simplified version)

A task status (simplified version) is referenced by issuing the following service call from the processing program.

- ref_tst, iref_tst
  Stores task state packet (current state, reason for waiting) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtst*.
  Used for referencing only the current state and reason for wait among task information.
  Response becomes faster than using ref_tsk or iref_tsk because only a few information items are acquired.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = 8;              /*Declares and initializes variable*/
    T_RTST  pk_rtst;                /*Declares data structure*/
    STAT    tskstat;                /*Declares variable*/
    STAT    tskwait;                /*Declares variable*/

    .........
    .........

    ref_tst (tskid, &pk_rtst);      /*Reference task state (simplified version)*/

    tskstat = pk_rtst.tskstat;      /*Reference current state*/
    tskwait = pk_rtst.tskwait;      /*Reference reason for waiting*/

    .........
    .........
}
```

Note For details about the task state packet (simplified version), refer to "15.2.2 Task state packet (simplified version)".

3.10 Memory Saving

The RI850V4 provides the method (Disable preempt) for reducing the task stack size required by tasks to perform processing.

3.10.1 Disable preempt

In the RI850V4, preempt acknowledge status attribute TA_DISPREEMPT can be defined in Task information when creating a system configuration file.

The task for which this attribute is defined performs the operation that continues processing by ignoring the scheduling request issued from a non-task, so a management area of 24 to 44 bytes can be reduced per task.

CHAPTER 4 TASK DEPENDENT SYNCHRONIZATION FUNCTIONS

This chapter describes the task dependent synchronization functions performed by the RI850V4.

4.1 Outline

The RI850V4 provides several task-dependent synchronization functions.

4.2 Put Task to Sleep

4.2.1 Waiting forever

A task is moved to the sleeping state (waiting forever) by issuing the following service call from the processing program.

- slp_tsk
  As a result, the invoking task is unlinked from the ready queue and excluded from the RI850V4 scheduling subject.
  If a wakeup request has been queued to the target task (the wakeup request counter is not set to 0x0) when this service call is issued, this service call does not move the state but decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter).
  The sleeping state is cancelled in the following cases, and then moved to the READY state.

Sleeping State Cancel Operation	Return Value
A wakeup request was issued as a result of issuing wup_tsk.	E_OK
A wakeup request was issued as a result of issuing iwup_tsk.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/

    .........
    .........

    ercd = slp_tsk ();              /*Put task to sleep (waiting forever)*/

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

4.2.2 With timeout

A task is moved to the sleeping state (with timeout) by issuing the following service call from the processing program.

- tslp_tsk
  This service call moves an invoking task from the RUNNING state to the WAITING state (sleeping state).
  As a result, the invoking task is unlinked from the ready queue and excluded from the RI850V4 scheduling subject.
  If a wakeup request has been queued to the target task (the wakeup request counter is not set to 0x0) when this service call is issued, this service call does not move the state but decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter).
  The sleeping state is cancelled in the following cases, and then moved to the READY state.

Sleeping State Cancel Operation	Return Value
A wakeup request was issued as a result of issuing wup_tsk.	E_OK
A wakeup request was issued as a result of issuing iwup_tsk.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;               /*Declares variable*/
    TMO     tmout = 3600;       /*Declares and initializes variable*/

    .........
    .........

    ercd = tslp_tsk (tmout);    /*Put task to sleep (with timeout)*/

    if (ercd == E_OK) {
        .........               /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........               /*Forced termination processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........               /*Timeout processing*/
        .........
    }

    .........
    .........
}
```

Note When TMO_FEVR is specified for wait time *tmout*, processing equivalent to slp_tsk will be executed.

4.3 Wakeup Task

A task is woken up by issuing the following service call from the processing program.

- wup_tsk, iwup_tsk
  These service calls cancel the WAITING state (sleeping state) of the task specified by parameter *tskid*.
  As a result, the target task is moved from the sleeping state to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  If the target task is in a state other than the sleeping state when this service call is issued, this service call does not move the state but increments the wakeup request counter (by added 0x1 to the wakeup request counter).
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = ID_TSK1;        /*Declares and initializes variable*/

    .........
    .........

    wup_tsk (tskid);                /*Wakeup task*/

    .........
    .........
}
```

Note The wakeup request counter managed by the RI850V4 is configured in 7-bit widths. If the number of wakeup requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

4.4 Cancel Task Wakeup Requests

A wakeup request is cancelled by issuing the following service call from the processing program.

- can_wup, ican_wup
  These service calls cancel all of the wakeup requests queued to the task specified by parameter *tskid* (the wakeup request counter is set to 0x0).
  When this service call is terminated normally, the number of cancelled wakeup requests is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER_UINT ercd;                   /*Declares variable*/
    ID      tskid = ID_TSK1;        /*Declares and initializes variable*/

    .........
    .........

    ercd = can_wup (tskid);         /*Cancel task wakeup requests*/

    if (ercd >= 0x0) {
        .........                   /*Normal termination processing*/
        .........
    }

    .........
    .........
}
```

4.5 Release Task from Waiting

The WAITING state is forcibly cancelled by issuing the following service call from the processing program.

- rel_wai, irel_wai
  These service calls forcibly cancel the WAITING state of the task specified by parameter *tskid*.
  As a result, the target task unlinked from the wait queue and is moved from the WAITING state to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  "E_RLWAI" is returned from the service call that triggered the move to the WAITING state (slp_tsk, wai_sem, or the like) to the task whose WAITING state is cancelled by this service call.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = ID_TSK1;        /*Declares and initializes variable*/

    .........
    .........

    rel_wai (tskid);                /*Release task from waiting*/

    .........
    .........
}
```

Note 1 This service call does not perform queuing of forced cancellation requests. If the target task is in a state other than the WAITING or WAITING-SUSPENDED state, "E_OBJ" is returned.

Note 2 The SUSPENDED state is not cancelled by these service calls.

4.6 Suspend Task

A task is moved to the SUSPENDED state by issuing the following service call from the processing program.

- sus_tsk, isus_tsk
  These service calls add 0x1 to the suspend request counter for the task specified by parameter *tskid*, and then move the target task from the RUNNING state to the SUSPENDED state, from the READY state to the SUSPENDED state, or from the WAITING state to the WAITING-SUSPENDED state.
  If the target task has moved to the SUSPENDED or WAITING-SUSPENDED state when this service call is issued, the counter manipulation processing is not performed but only the suspend request counter increment processing is executed.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = ID_TSK1;        /*Declares and initializes variable*/

    .........
    .........

    sus_tsk (tskid);                /*Suspend task*/

    .........
    .........
}
```

Note The suspend request counter managed by the RI850V4 is configured in 7-bit widths. If the number of suspend requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

4.7 Resume Suspended Task

4.7.1 Resume suspended task

The SUSPENDED state is cancelled by issuing the following service call from the processing program.

- rsm_tsk, irsm_tsk
  This service call subtracts 0x1 from the suspend request counter for the task specified by parameter *tskid*, and then cancels the SUSPENDED state of the target task.
  As a result, the target task is moved from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.
  If a suspend request is queued (subtraction result is other than 0x0) when this service call is issued, the counter manipulation processing is not performed but only the suspend request counter decrement processing is executed.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = ID_TSK1;        /*Declares and initializes variable*/

    .........
    .........

    rsm_tsk (tskid);                /*Resume suspended task*/

    .........
    .........
}
```

Note This service call does not perform queuing of cancellation requests. If the target task is in a state other than the SUSPENDED or WAITING-SUSPENDED state, "E_OBJ" is therefore returned.

4.7.2 Forcibly resume suspended task

The SUSPENDED state is forcibly cancelled by issuing the following service calls from the processing program.

- frsm_tsk, ifrsm_tsk
  These service calls cancel all of the suspend requests issued for the task specified by parameter *tskid* (by setting the suspend request counter to 0x0). As a result, the target task moves from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      tskid = ID_TSK1;        /*Declares and initializes variable*/

    .........
    .........

    frsm_tsk (tskid);               /*Forcibly resume suspended task*/

    .........
    .........
}
```

Note This service call does not perform queuing of cancellation requests. If the target task is in a state other than the SUSPENDED or WAITING-SUSPENDED state, "E_OBJ" is therefore returned.

4.8 Delay Task

A task is moved to the delayed state by issuing the following service call from the processing program.

- dly_tsk
  This service call moves the invoking task from the RUNNING state to the WAITING state (delayed state).
  As a result, the invoking task is unlinked from the ready queue and excluded from the RI850V4 scheduling subject.
  The delayed state is cancelled in the following cases, and then moved to the READY state.

Delayed State Cancel Operation	Return Value
Delay time specified by parameter *dlytim* has elapsed.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    RELTIM  dlytim = 3600;          /*Declares and initializes variable*/

    .........
    .........

    ercd = dly_tsk (dlytim);        /*Delay task*/

    if (ercd == E_OK) {
        .........              /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........              /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

4.9 Differences Between Wakeup Wait with Timeout and Time Elapse Wait

Wakeup waits with timeout and time elapse waits differ on the following points.

Table 4-1 Differences Between Wakeup Wait with Timeout and Time Elapse Wait

	Wakeup Wait with Timeout	Time Elapse Wait
Service call that causes status change	tslp_tsk	dly_tsk
Return value when timed out	E_TMOUT	E_OK
Operation when wup_tsk or iwup_tsk is issued	Wakeup	Queues the wakeup request (time elapse wait is not cancelled).

CHAPTER 5 SYNCHRONIZATION AND COMMUNICATION FUNCTIONS

This chapter describes the synchronization and communication functions performed by the RI850V4.

5.1 Outline

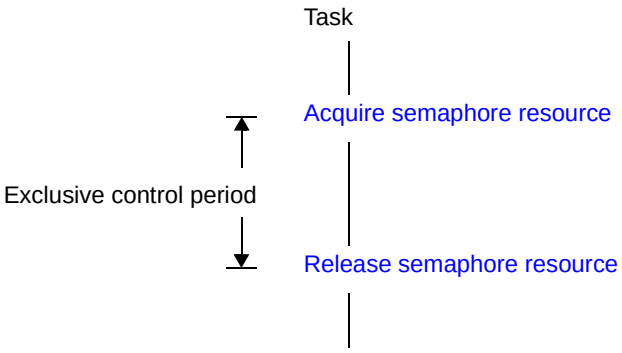
The synchronization and communication functions of the RI850V4 consist of Semaphores, Eventflags, Data Queues, and Mailboxes that are provided as means for realizing exclusive control, queuing, and communication among tasks.

5.2 Semaphores

In the RI850V4, non-negative number counting semaphores are provided as a means (exclusive control function) for preventing contention for limited resources (hardware devices, library function, etc.) arising from the required conditions of simultaneously running tasks.

The following shows a processing flow when using a semaphore.

Figure 5-1 Processing Flow (Semaphore)



5.2.1 Create semaphore

In the RI850V4, the method of creating a semaphore is limited to "static creation".

Semaphores therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static semaphore creation means defining of semaphores using static API "CRE_SEM" in the system configuration file.

For details about the static API "CRE_SEM", refer to "17.5.2 Semaphore information".

5.2.2 Acquire semaphore resource

A resource is acquired (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- wai_sem
  This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).
  If no resources are acquired from the target semaphore when this service call is issued (no available resources exist), this service call does not acquire resources but queues the invoking task to the target semaphore wait queue and moves it from the RUNNING state to the WAITING state (resource acquisition wait state).
  The WAITING state for a semaphore resource is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Semaphore Resource Cancel Operation	Return Value
The resource was returned to the target semaphore as a result of issuing sig_sem.	E_OK
The resource was returned to the target semaphore as a result of issuing isig_sem.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following describes an example for coding this service call.

```
#include    <kernel.h>          /*Standard header file definition*/
#include    <kernel_id.h>       /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;               /*Declares variable*/
    ID      semid = ID_SEM1;    /*Declares and initializes variable*/

    .........
    .........

    ercd = wai_sem (semid);     /*Acquire semaphore resource (waiting forever)*/

    if (ercd == E_OK) {
        .........               /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........               /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

Note Invoking tasks are queued to the target semaphore wait queue in the order defined during configuration (FIFO order or priority order).

- pol_sem, ipol_sem
  This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).
  If a resource could not be acquired from the target semaphore (semaphore counter is set to 0x0) when this service call is issued, the counter manipulation processing is not performed but "E_TMOUT" is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      semid = ID_SEM1;        /*Declares and initializes variable*/

    .........
    .........

    ercd = pol_sem (semid);         /*Acquire semaphore resource (polling)*/

    if (ercd == E_OK) {
        .........                   /*Polling success processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Polling failure processing*/
        .........
    }

    .........
    .........
}
```

- twai_sem
  This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).
  If no resources are acquired from the target semaphore when service call is issued this (no available resources exist), this service call does not acquire resources but queues the invoking task to the target semaphore wait queue and moves it from the RUNNING state to the WAITING state with timeout (resource acquisition wait state).
  The WAITING state for a semaphore resource is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Semaphore Resource Cancel Operation	Return Value
The resource was returned to the target semaphore as a result of issuing sig_sem.	E_OK
The resource was returned to the target semaphore as a result of issuing isig_sem.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      semid = ID_SEM1;        /*Declares and initializes variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

    .........
    .........

    ercd = twai_sem (semid, tmout); /*Acquire semaphore resource (with timeout)*/

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........              /*Timeout processing*/
        .........
    }

    .........
    .........
}
```

Note 1 Invoking tasks are queued to the target semaphore wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 TMO_FEVR is specified for wait time *tmout*, processing equivalent to wai_sem will be executed. When TMO_POL is specified, processing equivalent to pol_sem /ipol_sem will be executed.

5.2.3 Release semaphore resource

A resource is returned by issuing the following service call from the processing program.

- sig_sem, isig_sem
  These service calls return the resource to the semaphore specified by parameter *semid* (adds 0x1 to the semaphore counter).
  If a task is queued in the wait queue of the target semaphore when this service call is issued, the counter manipulation processing is not performed but the resource is passed to the relevant task (first task of wait queue).
  As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (WAITING state for a semaphore resource) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      semid = ID_SEM1;        /*Declares and initializes variable*/

    .........
    .........

    sig_sem (semid);                /*Release semaphore resource*/

    .........
    .........
}
```

Note With the RI850V4, the maximum possible number of semaphore resources (maximum resource count) is defined during configuration. If the number of resources exceeds the specified maximum resource count, this service call therefore does not return the acquired resources (addition to the semaphore counter value) but returns E_QOVR.

5.2.4 Reference semaphore state

A semaphore status is referenced by issuing the following service call from the processing program.

- ref_sem, iref_sem
  Stores semaphore state packet (ID number of the task at the head of the wait queue, current resource count, etc.) of the semaphore specified by parameter *semid* in the area specified by parameter *pk_rsem*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      semid = ID_SEM1;        /*Declares and initializes variable*/
    T_RSEM  pk_rsem;                /*Declares data structure*/
    ID      wtskid;                 /*Declares variable*/
    UINT    semcnt;                 /*Declares variable*/
    ATR     sematr;                 /*Declares variable*/
    UINT    maxsem;                 /*Declares variable*/

    .........
    .........

    ref_sem (semid, &pk_rsem);      /*Reference semaphore state*/

    wtskid = pk_rsem.wtskid;        /*Reference ID number of the task at the */
                                    /*head of the wait queue*/
    semcnt = pk_rsem.semcnt;        /*Reference current resource count*/
    sematr = pk_rsem.sematr;        /*Reference attribute*/
    maxsem = pk_rsem.maxsem;        /*Reference maximum resource count*/

    .........
    .........
}
```

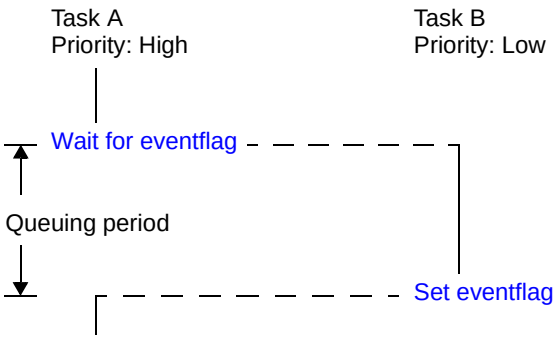
Note For details about the semaphore state packet, refer to "15.2.3 Semaphore state packet".

5.3 Eventflags

The RI850V4 provides 32-bit eventflags as a queuing function for tasks, such as keeping the tasks waiting for execution, until the results of the execution of a given processing program are output.

The following shows a processing flow when using an eventflag.

Figure 5-2 Processing Flow (Eventflag)



5.3.1 Create eventflag

In the RI850V4, the method of creating an eventflag is limited to "static creation".

Eventflags therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static event flag creation means defining of event flags using static API "CRE_FLG" in the system configuration file.

For details about the static API "CRE_FLG", refer to "17.5.3 Eventflag information".

5.3.2 Set eventflag

Bit pattern is set by issuing the following service call from the processing program.

- set_flg, iset_flg
  These service calls set the result of logical OR operating the bit pattern of the eventflag specified by parameter *flgid* and the bit pattern specified by parameter *setptn* as the bit pattern of the target eventflag.
  If the required condition of the task queued to the target eventflag wait queue is satisfied when this service call is issued, the relevant task is unlinked from the wait queue at the same time as bit pattern setting processing.
  As a result, the relevant task is moved from the WAITING state (WAITING state for an eventflag) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      flgid = ID_FLG1;        /*Declares and initializes variable*/
    FLGPTN  setptn = 10;            /*Declares and initializes variable*/

    .........
    .........

    set_flg (flgid, setptn);        /*Set eventflag*/

    .........
    .........
}
```

Note 1 If the bit pattern set to the target eventflag is B'1100 and the bit pattern specified by parameter *setptn* is B'1010 when this service call is issued, the bit pattern of the target eventflag is set to B'1110.

Note 2 When the TA_WMUL attribute is specified for the target eventflag, the range of tasks to be checked on "whether issuing of this service call satisfies the required condition" differs depending on whether the TA_CLR attribute is also specified.

- When TA_CLR is specified
  Check begins from the task at the head of the wait queue and stops at the first task whose required condition is satisfied.
- When TA_CLR is not specified
  All tasks placed in the wait queue are checked.

5.3.3 Clear eventflag

A bit pattern is cleared by issuing the following service call from the processing program.

- clr_flg, iclr_flg
  This service call sets the result of logical AND operating the bit pattern set to the eventflag specified by parameter *flgid* and the bit pattern specified by parameter *clrptn* as the bit pattern of the target eventflag.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      flgid = ID_FLG1;        /*Declares and initializes variable*/
    FLGPTN  clrptn = 10;            /*Declares and initializes variable*/

    .........
    .........

    clr_flg (flgid, clrptn);        /*Clear eventflag*/

    .........
    .........
}
```

Note If the bit pattern set to the target eventflag is B'1100 and the bit pattern specified by parameter *clrptn* is B'1010 when this service call is issued, the bit pattern of the target eventflag is set to B'1110.

5.3.4 Wait for eventflag

A bit pattern is checked (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- wai_flg
  This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.
  If a bit pattern that satisfies the required condition has been set for the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.
  If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, the invoking task is queued to the target eventflag wait queue.
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (WAITING state for an eventflag).
  The WAITING state for an eventflag is cancelled in the following cases, and then moved to the READY state.

WAITING State for an Eventflag Cancel Operation	Return Value
A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing set_flg.	E_OK
A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing iset_flg.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.
- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      flgid = ID_FLG1;        /*Declares and initializes variable*/
    FLGPTN  waiptn = 14;            /*Declares and initializes variable*/
    MODE    wfmode = TWF_ANDW;      /*Declares and initializes variable*/
    FLGPTN  p_flgptn;               /*Declares variable*/

    .........
    .........

                                    /*Wait for eventflag (waiting forever)*/
    ercd = wai_flg (flgid, waiptn, wfmode, &p_flgptn);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }
```

```
    .........
    .........
}
```

Note 1 With the RI850V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TW_WSGL attribute) to which a wait task is queued, therefore, "E_ILUSE" is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL:	Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL:	Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 Invoking tasks are queued to the target event flag (TA_WMUL attribute) wait queue in the order defined during configuration (FIFO order or priority order).

Note 3 The RI850V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 4 If the WAITING state for an eventflag is forcibly released by issuing rel_wai or irel_wai, the contents of the area specified by parameter *p_flgptn* will be undefined.

- pol_flg, ipol_flg
  This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.
  If the bit pattern that satisfies the required condition has been set to the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.
  If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, "E_TMOUT" is returned.
  The following shows the specification format of required condition *wfmode*.

  - *wfmode* = TWF_ANDW
    Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.
  - *wfmode* = TWF_ORW
    Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      flgid = ID_FLG1;        /*Declares and initializes variable*/
    FLGPTN  waiptn = 14;            /*Declares and initializes variable*/
    MODE    wfmode = TWF_ANDW;      /*Declares and initializes variable*/
    FLGPTN  p_flgptn;               /*Declares variable*/

    .........
    .........

                                    /*Wait for eventflag (polling)*/
    ercd = pol_flg (flgid, waiptn, wfmode, &p_flgptn);

    if (ercd == E_OK) {
        .........              /*Polling success processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........              /*Polling failure processing*/
        .........
    }

    .........
    .........
}
```

Note 1 With the RI850V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TW_WSGL attribute) to which a wait task is queued, therefore, "E_ILUSE" is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL:	Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL:	Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 The RI850V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 3 If the bit pattern of the target event flag does not satisfy the required condition when this service call is issued, the contents in the area specified by parameter *p_flgptn* become undefined.

- twai_flg
  This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.
  If a bit pattern that satisfies the required condition has been set for the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.
  If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, the invoking task is queued to the target eventflag wait queue.
  As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (WAITING state for an eventflag).
  The WAITING state for an eventflag is cancelled in the following cases, and then moved to the READY state.

WAITING State for an Eventflag Cancel Operation	Return Value
A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing set_flg.	E_OK
A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing iset_flg.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.
- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      flgid = ID_FLG1;        /*Declares and initializes variable*/
    FLGPTN  waiptn = 14;            /*Declares and initializes variable*/
    MODE    wfmode = TWF_ANDW;      /*Declares and initializes variable*/
    FLGPTN  p_flgptn;               /*Declares variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

    .........
    .........

                                    /*Wait for eventflag (with timeout)*/
    ercd = twai_flg (flgid, waiptn, wfmode, &p_flgptn, tmout);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Timeout processing*/
        .........
    }

    .........
    .........
}
}
```

Note 1 With the RI850V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TW_WSGL attribute) to which a wait task is queued, therefore, "E_ILUSE" is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL:	Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL:	Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 Invoking tasks are queued to the target event flag (TA_WMUL attribute) wait queue in the order defined during configuration (FIFO order or priority order).

Note 3 The RI850V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 4 If the event flag wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_flgptn* become undefined.

Note 5 TMO_FEVR is specified for wait time *tmout*, processing equivalent to wai_flg will be executed. When TMO_POL is specified, processing equivalent to pol_flg /ipol_flg will be executed.

5.3.5 Reference eventflag state

An eventflag status is referenced by issuing the following service call from the processing program.

- ref_flg, iref_flg
  Stores eventflag state packet (ID number of the task at the head of the wait queue, current bit pattern, etc.) of the eventflag specified by parameter *flgid* in the area specified by parameter *pk_rflg*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      flgid = ID_FLG1;        /*Declares and initializes variable*/
    T_RFLG  pk_rflg;                /*Declares data structure*/
    ID      wtskid;                 /*Declares variable*/
    FLGPTN  flgptn;                 /*Declares variable*/
    ATR     flgatr;                 /*Declares variable*/

    .........
    .........

    ref_flg (flgid, &pk_rflg);      /*Reference eventflag state*/

    wtskid = pk_rflg.wtskid;        /*Reference ID number of the task at the */
                                    /*head of the wait queue*/
    flgptn = pk_rflg.flgptn;        /*Reference current bit pattern*/
    flgatr = pk_rflg.flgatr;        /*Reference attribute*/

    .........
    .........
}
```

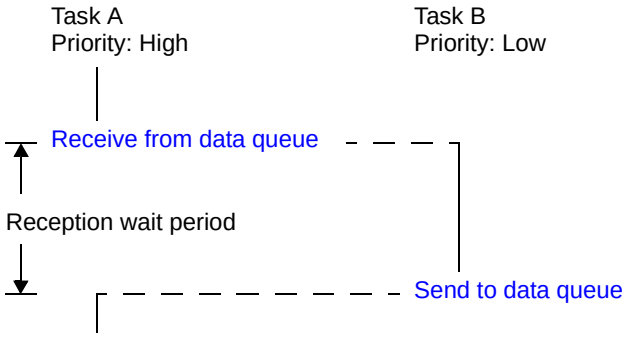
Note For details about the eventflag state packet, refer to "15.2.4 Eventflag state packet".

5.4 Data Queues

Multitask processing requires the inter-task communication function (data transfer function) that reports the processing result of a task to another task. The RI850V4 therefore provides the data queues that have the data queue area in which data read/write is enabled for transferring the prescribed size of data.

The following shows a processing flow when using a data queue.

Figure 5-3 Processing Flow (Data Queue)



Note Data units of 4 bytes are transmitted or received at a time.

5.4.1 Create data queue

In the RI850V4, the method of creating a data queue is limited to "static creation".

Data queues therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static data queue creation means defining of data queues using static API "CRE_DTQ" in the system configuration file.

For details about the static API "CRE_DTQ", refer to "17.5.4 Data queue information".

5.4.2 Send to data queue

A data is transmitted by issuing the following service call from the processing program.

- snd_dtq
  This service call writes data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.
  If there is no available space for writing data in the data queue area of the target data queue when this service call is issued, this service call does not write data but queues the invoking task to the transmission wait queue of the target data queue and moves it from the RUNNING state to the WAITING state (data transmission wait state).
  The sending WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

Sending WAITING State for a Data Queue Cancel Operation	Return Value
Available space was secured in the data queue area of the target data queue as a result of issuing rcv_dtq.	E_OK
Available space was secured in the data queue area of the target data queue as a result of issuing prcv_dtq.	E_OK
Available space was secured in the data queue area of the target data queue as a result of issuing iprcv_dtq.	E_OK
Available space was secured in the data queue area of the target data queue as a result of issuing trcv_dtq.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      dtqid = ID_DTQ1;        /*Declares and initializes variable*/
    VP_INT  data = 123;             /*Declares and initializes variable*/

    .........
    .........

    ercd = snd_dtq (dtqid, data);   /*Send to data queue (waiting forever)*/

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

Note 1 Data is written to the data queue area of the target data queue in the order of the data transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target data queue in the order defined during configuration (FIFO order or priority order).

- psnd_dtq, ipsnd_dtq

These service calls write data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.
If there is no available space for writing data in the data queue area of the target data queue when either of these service calls is issued, data is not written but E_TMOUT is returned.
If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      dtqid = ID_DTQ1;        /*Declares and initializes variable*/
    VP_INT  data = 123;             /*Declares and initializes variable*/

    .........
    .........

                                    /*Send to data queue (polling)*/
    ercd = psnd_dtq (dtqid, data);

    if (ercd == E_OK) {
        .........                   /*Polling success processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Polling failure processing*/
        .........
    }

    .........
    .........
}
```

Note Data is written to the data queue area of the target data queue in the order of the data transmission request.

- tsnd_dtq
  This service call writes data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.
  If there is no available space for writing data in the data queue area of the target data queue when this service call is issued, the service call does not write data but queues the invoking task to the transmission wait queue of the target data queue and moves it from the RUNNING state to the WAITING state with time (data transmission wait state).
  The sending WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

Sending WAITING State for a Data Queue Cancel Operation	Return Value
An available space was secured in the data queue area of the target data queue as a result of issuing rcv_dtq.	E_OK
An available space was secured in the data queue area of the target data queue as a result of issuing prcv_dtq.	E_OK
An available space was secured in the data queue area of the target data queue as a result of issuing iprcv_dtq.	E_OK
An available space was secured in the data queue area of the target data queue as a result of issuing trcv_dtq.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

  If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      dtqid = ID_DTQ1;        /*Declares and initializes variable*/
    VP_INT  data = 123;             /*Declares and initializes variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

    .........
    .........

                                    /*Send to data queue (with timeout)*/
    ercd = tsnd_dtq (dtqid, data, tmout);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Timeout processing*/
        .........
    }

    .........
    .........
}
```

Note 1 Data is written to the data queue area of the target data queue in the order of the data transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target data queue in the order defined during configuration (FIFO order or priority order).

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to snd_dtq will be executed. When TMO_POL is specified, processing equivalent to psnd_dtq /ipsnd_dtq will be executed.

5.4.3 Forced send to data queue

Data is forcibly transmitted by issuing the following service call from the processing program.

- fsnd_dtq, ifsnd_dtq
  These service calls write data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.
  If there is no available space for writing data in the data queue area of the target data queue when either of these service calls is issued, the service call overwrites data to the area with the oldest data that was written.
  If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      dtqid = ID_DTQ1;        /*Declares and initializes variable*/
    VP_INT  data = 123;             /*Declares and initializes variable*/

    .........
    .........

    fsnd_dtq (dtqid, data);         /*Forced send to data queue*/

    .........
    .........
}
```

5.4.4 Receive from data queue

A data is received (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- rcv_dtq
  This service call reads data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.
  If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when this service call is issued, the service call does not read data but queues the invoking task to the reception wait queue of the target data queue and moves it from the RUNNING state to the WAITING state (data reception wait state).
  The receiving WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

Receiving WAITING State for a Data Queue Cancel Operation	Return Value
Data was written to the data queue area of the target data queue as a result of issuing snd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing psnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing ipsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing tsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing fsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing ifsnd_dtq.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      dtqid = ID_DTQ1;        /*Declares and initializes variable*/
    VP_INT  p_data;                 /*Declares variable*/

    .........
    .........

                                    /*Receive from data queue (waiting forever)*/
    ercd = rcv_dtq (dtqid, &p_data);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }
```

```
    .........
    .........
}
```

Note 1 Invoking tasks are queued to the reception wait queue of the target data queue in the order of the data reception request.

Note 2 If the receiving WAITING state for a data queue is forcibly released by issuing rel_wai or irel_wai, the contents of the area specified by parameter *p_data* will be undefined.

- prcv_dtq, iprcv_dtq

These service calls read data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.
If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when either of these service calls is issued, the service call does not read data but E_TMOUT is returned.
The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      dtqid = ID_DTQ1;        /*Declares and initializes variable*/
    VP_INT  p_data;                 /*Declares variable*/

    .........
    .........

                                    /*Receive from data queue (polling)*/
    ercd = prcv_dtq (dtqid, &p_data);

    if (ercd == E_OK) {
        .........                   /*Polling success processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Polling failure processing*/
        .........
    }

    .........
    .........
}
```

Note If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when either of these service calls is issued, the contents in the area specified by parameter *p_data* become undefined.

- trcv_dtq
  This service call reads data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.
  If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when this service call is issued, the service call does not read data but queues the invoking task to the reception wait queue of the target data queue and moves it from the RUNNING state to the WAITING state with time out (data reception wait state).
  The receiving WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

Receiving WAITING State for a Data Queue Cancel Operation	Return Value
Data was written to the data queue area of the target data queue as a result of issuing snd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing psnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing ipsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing tsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing fsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing ifsnd_dtq.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      dtqid = ID_DTQ1;        /*Declares and initializes variable*/
    VP_INT  p_data;                 /*Declares variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

    .........
    .........

                                    /*Receive from data queue (with timeout)*/
    ercd = trcv_dtq (dtqid, &p_data, tmout);

    if (ercd == E_OK) {
            .........               /*Normal termination processing*/
            .........
    } else if (ercd == E_RLWAI) {
            .........               /*Forced termination processing*/
            .........
    } else if (ercd == E_TMOUT) {
            .........               /*Timeout processing*/
            .........
    }
    .........
    .........
}
```

Note 1 Invoking tasks are queued to the reception wait queue of the target data queue in the order of the data reception request.

Note 2 If the data reception wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_data* become undefined.

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to rcv_dtq will be executed. When TMO_POL is specified, processing equivalent to prcv_dtq /iprcv_dtq will be executed.

5.4.5 Reference data queue state

A data queue status is referenced by issuing the following service call from the processing program.

- ref_dtq, iref_dtq
  These service calls store the detailed information of the data queue (existence of waiting tasks, number of data elements in the data queue, etc.) specified by parameter *dtqid* into the area specified by parameter *pk_rdtq*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      dtqid = ID_DTQ1;        /*Declares and initializes variable*/
    T_RDTQ  pk_rdtq;                /*Declares data structure*/
    ID      stskid;                 /*Declares variable*/
    ID      rtskid;                 /*Declares variable*/
    UINT    sdtqcnt;                /*Declares variable*/
    ATR     dtqatr;                 /*Declares variable*/
    UINT    dtqcnt;                 /*Declares variable*/

    .........
    .........

    ref_dtq (dtqid, &pk_rdtq);      /*Reference data queue state*/

    stskid = pk_rdtq.stskid;        /*Acquires existence of tasks waiting for */
                                    /*data transmission*/
    rtskid = pk_rdtq.rtskid;        /*Acquires existence of tasks waiting for */
                                    /*data reception*/
    sdtqcnt = pk_rdtq.sdtqcnt;      /*Reference the number of data elements in */
                                    /*data queue*/
    dtqatr = pk_rdtq.dtqatr;        /*Reference attribute*/
    dtqcnt = pk_rdtq.dtqcnt;        /*Reference data count*/

    .........
    .........
}
```

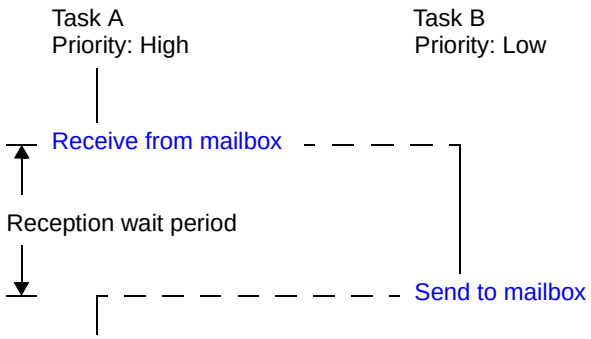
Note For details about the data queue state packet, refer to "15.2.5 Data queue state packet".

5.5 Mailboxes

The RI850V4 provides a mailbox, as a communication function between tasks, that hands over the execution result of a given processing program to another processing program.

The following shows a processing flow when using a mailbox.

Figure 5-4 Processing Flow (Mailbox)



5.5.1 Messages

The information exchanged among processing programs via the mailbox is called "messages".

Messages can be transmitted to any processing program via the mailbox, but it should be noted that, in the case of the synchronization and communication functions of the RI850V4, only the start address of the message is handed over to the receiving processing program, but the message contents are not copied to a separate area.

- Securement of memory area
  In the case of the RI850V4, it is recommended to use the memory area secured by issuing service calls such as get_mpf and get_mpl for messages.

  Note The RI850V4 uses the message start area as a link area during queuing to the wait queue for mailbox messages. Therefore, if the memory area for messages is secured from other than the memory area controlled by the RI850V4, it must be secured from 4-byte aligned addresses.

- Basic form of messages
  In the RI850V4, the message contents and length are prescribed as follows, according to the attributes of the mailbox to be used.

  - When using a mailbox with the TA_MFIFO attribute
    The contents and length past the first 4 bytes of a message (system reserved area msgnext) are not restricted in particular in the RI850V4.
    Therefore, the contents and length past the first 4 bytes are prescribed among the processing programs that exchange data using the mailbox with the TA_MFIFO attribute.
    The following shows the basic form of coding TA_MFIFO attribute messages in C.

    [Message packet for TA_MFIFO attribute]

```
typedef struct  t_msg {
    struct  t_msg   *msgnext;       /*Reserved for future use*/
} T_MSG;
```

- When using a mailbox with the TA_MPRI attribute
  The contents and length past the first 8 bytes of a message (system reserved area msgque, priority level msgpri) are not restricted in particular in the RI850V4.
  Therefore, the contents and length past the first 8 bytes are prescribed among the processing programs that exchange data using the mailbox with the TA_MPRI attribute.
  The following shows the basic form of coding TA_MPRI attribute messages in C.

  [Message packet for TA_MPRI attribute]

```
typedef struct  t_msg_pri {
    struct  t_msg   msgque;         /*Reserved for future use*/
    PRI     msgpri;                 /*Message priority*/
} T_MSG_PRI;
```

Note 1 In the RI850V4, a message having a smaller priority number is given a higher priority.

Note 2 Values that can be specified as the message priority level are limited to the range defined in Mailbox information (Maximum message priority: maxmpri) when the system configuration file is created.

Note 3 For details about the message packet, refer to "15.2.6 Message packet".

5.5.2 Create mailbox

In the RI850V4, the method of creating a mailbox is limited to "static creation".

Mailboxes therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static mailbox creation means defining of mailboxes using static API "CRE_MBX" in the system configuration file.

For details about the static API "CRE_MBX", refer to "17.5.5 Mailbox information".

5.5.3 Send to mailbox

A message is transmitted by issuing the following service call from the processing program.

- snd_mbx, isnd_mbx
  This service call transmits the message specified by parameter *pk_msg* to the mailbox specified by parameter *mbxid* (queues the message in the wait queue).
  If a task is queued to the target mailbox wait queue when this service call is issued, the message is not queued but handed over to the relevant task (first task of the wait queue).
  As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (receiving WAITING state for a mailbox) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      mbxid = ID_MBX1;        /*Declares and initializes variable*/
    T_MSG_PRI       *pk_msg;        /*Declares data structure*/

    .........
    .........

    .........                       /*Secures memory area (for message)*/
    .........

    .........                       /*Creates message (contents)*/
    .........

    pk_msg->msgpri = 8;             /*Initializes data structure*/

                                    /*Send to mailbox*/
    snd_mbx (mbxid, (T_MSG *) pk_msg);

    .........
    .........
}
```

Note 1 Messages are queued to the target mailbox wait queue in the order defined by queuing method during configuration (FIFO order or priority order).

Note 2 With the RI850V4 mailbox, only the start address of the message is handed over to the receiving processing program, but the message contents are not copied to a separate area. The message contents can therefore be rewritten even after this service call is issued.

Note 3 For details about the message packet, refer to "15.2.6 Message packet".

5.5.4 Receive from mailbox

A message is received (infinite wait, polling, or with timeout) by issuing the following service call from the processing program.

- rcv_mbx
  This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.
  If no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, this service call does not receive messages but queues the invoking task to the target mailbox wait queue and moves it from the RUNNING state to the WAITING state (message reception wait state).
  The receiving WAITING state for a mailbox is cancelled in the following cases, and then moved to the READY state.

Receiving WAITING State for a Mailbox Cancel Operation	Return Value
A message was transmitted to the target mailbox as a result of issuing snd_mbx.	E_OK
A message was transmitted to the target mailbox as a result of issuing isnd_mbx.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbxid = ID_MBX1;        /*Declares and initializes variable*/
    T_MSG   *ppk_msg;               /*Declares data structure*/

    .........
    .........

                                    /*Receive from mailbox*/
    ercd = rcv_mbx (mbxid, &ppk_msg);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

Note 1 Invoking tasks are queued to the target mailbox wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 If the receiving WAITING state for a mailbox is forcibly released by issuing rel_wai or irel_wai, the contents of the area specified by parameter *ppk_msg* will be undefined.

Note 3 For details about the message packet, refer to "15.2.6 Message packet".

- prcv_mbx, iprcv_mbx
  This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.
  If the message could not be received from the target mailbox (no messages were queued in the wait queue) when this service call is issued, message reception processing is not executed but "E_TMOUT" is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbxid = ID_MBX1;        /*Declares and initializes variable*/
    T_MSG   *ppk_msg;               /*Declares data structure*/

    .........
    .........

                                    /*Receive from mailbox (polling)*/
    ercd = prcv_mbx (mbxid, &ppk_msg);

    if (ercd == E_OK) {
        .........                   /*Polling success processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Polling failure processing*/
        .........
    }

    .........
    .........
}
```

Note 1 If no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, the contents in the area specified by parameter *ppk_msg* become undefined.

Note 2 For details about the message packet, refer to "15.2.6 Message packet".

- trcv_mbx

This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.
If no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, this service call does not receive messages but queues the invoking task to the target mailbox wait queue and moves it from the RUNNING state to the WAITING state with timeout (message reception wait state).
The receiving WAITING state for a mailbox is cancelled in the following cases, and then moved to the READY state.

Receiving WAITING State for a Mailbox Cancel Operation	Return Value
A message was transmitted to the target mailbox as a result of issuing snd_mbx.	E_OK
A message was transmitted to the target mailbox as a result of issuing isnd_mbx.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mbxid = ID_MAX1;        /*Declares and initializes variable*/
    T_MSG   *ppk_msg;               /*Declares data structure*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

    .........
    .........

                                    /*Receive from mailbox (with timeout)*/
    ercd = trcv_mbx (mbxid, &ppk_msg, tmout);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Timeout processing*/
        .........
    }

    .........
    .........
}
```

Note 1 Invoking tasks are queued to the target mailbox wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 If the message reception wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *ppk_msg* become undefined.

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to rcv_mbx will be executed. When TMO_POL is specified, processing equivalent to prcv_mbx /iprcv_mbx will be executed.

Note 4 For details about the message packet, refer to "15.2.6 Message packet".

5.5.5 Reference mailbox state

A mailbox status is referenced by issuing the following service call from the processing program.

- ref_mbx, iref_mbx
  Stores mailbox state packet (ID number of the task at the head of the wait queue, start address of the message packet at the head of the wait queue) of the mailbox specified by parameter *mbxid* in the area specified by parameter *pk_rmbx*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      mbxid = ID_MBX1;        /*Declares and initializes variable*/
    T_RMBX  pk_rmbx;                /*Declares data structure*/
    ID      wtskid;                 /*Declares variable*/
    T_MSG   *pk_msg;                /*Declares data structure*/
    ATR     mbxatr;                 /*Declares variable*/

    .........
    .........

    ref_mbx (mbxid, &pk_rmbx);      /*Reference mailbox state*/

    wtskid = pk_rmbx.wtskid;        /*Reference ID number of the task at the */
                                    /*head of the wait queue*/
    pk_msg = pk_rmbx.pk_msg;        /*Reference start address of the message */
                                    /*packet at the head of the wait queue*/
    mbxatr = pk_rmbx.mbxatr;        /*Reference attribute*/

    .........
    .........
}
```

Note For details about the mailbox state packet, refer to "15.2.7 Mailbox state packet".

CHAPTER 6 EXTENDED SYNCHRONIZATION AND COMMUNICATION FUNCTIONS

This chapter describes the extended synchronization and communication functions performed by the RI850V4.

6.1 Outline

The RI850V4 provides Mutexes as the extended synchronization and communication function for implementing exclusive control between tasks.

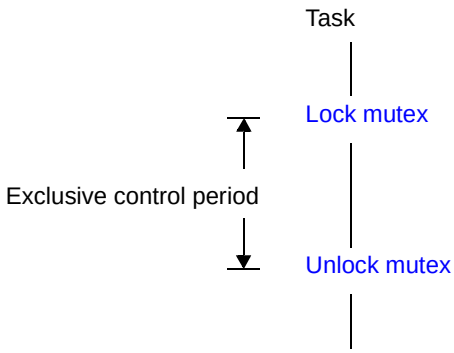
6.2 Mutexes

Multitask processing requires the function to prevent contentions on using the limited number of resources (A/D converter, coprocessor, files, or the like) simultaneously by tasks operating in parallel (exclusive control function). To resolve such problems, the RI850V4 therefore provides "mutexes".

The following shows a processing flow when using a mutex.

The mutexes provided in the RI850V4 do not support the priority inheritance protocol and priority ceiling protocol but only support the FIFO order and priority order.

Figure 6-1 Processing Flow (Mutex)



6.2.1 Differences from semaphores

Since the mutexes of the RI850V4 do not support the priority inheritance protocol and priority ceiling protocol, so it operates similarly to semaphores (binary semaphore) whose the maximum resource count is 1, but they differ in the following points.

- A locked mutex can be unlocked (equivalent to returning of resources) only by the task that locked the mutex
  --> Semaphores can return resources via any task and handler.

- Unlocking is automatically performed when a task that locked the mutex is terminated (ext_tsk or ter_tsk)
  --> Semaphores do not return resources automatically, so they end with resources acquired.

- Semaphores can manage multiple resources (the maximum resource count can be assigned), but the maximum number of resources assigned to a mutex is fixed to 1.

6.2.2 Create mutex

In the RI850V4, the method of creating a mutex is limited to "static creation".

Mutexes therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static mutex creation means defining of mutexes using static API "CRE_MTX" in the system configuration file.

For details about the static API "CRE_MTX", refer to "17.5.6 Mutex information".

6.2.3 Lock mutex

Mutexes can be locked by issuing the following service call from the processing program.

- loc_mtx
  This service call locks the mutex specified by parameter *mtxid*.
  If the target mutex could not be locked (another task has been locked) when this service call is issued, this service call queues the invoking task to the target mutex wait queue and moves it from the RUNNING state to the WAITING state (mutex wait state).
  The WAITING state for a mutex is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Mutex Cancel Operation	Return Value
The locked state of the target mutex was cancelled as a result of issuing unl_mtx.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ext_tsk.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ter_tsk.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mtxid = ID_MTX1;        /*Declares and initializes variable*/

    .........
    .........

    ercd = loc_mtx (mtxid);         /*Lock mutex (waiting forever)*/

    if (ercd == E_OK) {
        .........                   /*Locked state*/
        .........

        unl_mtx (mtxid);            /*Unlock mutex*/
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

Note 1 Invoking tasks are queued to the target mutex wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 In the RI850V4, E_ILUSE is returned if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

- ploc_mtx
  This service call locks the mutex specified by parameter *mtxid*.
  If the target mutex could not be locked (another task has been locked) when this service call is issued but E_TMOUT is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mtxid = ID_MTX1;        /*Declares and initializes variable*/

    .........
    .........

    ercd = ploc_mtx (mtxid);        /*Lock mutex (polling)*/

    if (ercd == E_OK) {
        .........                   /*Polling success processing*/
        .........

        unl_mtx (mtxid);            /*Unlock mutex*/
    } else if (ercd == E_TMOUT) {
        .........                   /*Polling failure processing*/
        .........
    }

    .........
    .........
}
```

Note In the RI850V4, E_ILUSE is returned if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

- tloc_mtx

This service call locks the mutex specified by parameter *mtxid*.
If the target mutex could not be locked (another task has been locked) when this service call is issued, this service call queues the invoking task to the target mutex wait queue and moves it from the RUNNING state to the WAITING state with timeout (mutex wait state).
The WAITING state for a mutex is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Mutex Cancel Operation	Return Value
The locked state of the target mutex was cancelled as a result of issuing unl_mtx.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ext_tsk.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ter_tsk.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mtxid = ID_MTX1;        /*Declares and initializes variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

    .........
    .........

    ercd = tloc_mtx (mtxid, tmout); /*Lock mutex (with timeout)*/

    if (ercd == E_OK) {
        .........                   /*Locked state*/
        .........

        unl_mtx (mtxid);            /*Unlock mutex*/
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Timeout processing*/
        .........
    }

    .........
    .........

}
```

Note 1 Invoking tasks are queued to the target mutex wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 In the RI850V4, E_ILUSE is returned if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to loc_mtx will be executed. When TMO_POL is specified, processing equivalent to ploc_mtx will be executed.

6.2.4 Unlock mutex

The mutex locked state can be cancelled by issuing the following service call from the processing program.

- unl_mtx
  This service call unlocks the locked mutex specified by parameter *mtxid*.
  If a task has been queued to the target mutex wait queue when this service call is issued, mutex lock processing is performed by the task (the first task in the wait queue) immediately after mutex unlock processing.
  As a result, the task is unlinked from the wait queue and moves from the WAITING state (mutex wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mtxid = ID_MTX1;              /*Declares and initializes variable*/

    .........
    .........

    ercd = loc_mtx (mtxid);         /*Lock mutex*/

    if (ercd == E_OK) {
        .........                   /*Locked state*/
        .........

        unl_mtx (mtxid);            /*Unlock mutex*/
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

Note A locked mutex can be unlocked only by the task that locked the mutex.
If this service call is issued for a mutex that was not locked by an invoking task, no processing is performed but E_ILUSE is returned.

6.2.5 Reference mutex state

A mutex status is referenced by issuing the following service call from the processing program.

- ref_mtx, iref_mtx
  The service calls store the detailed information of the mutex specified by parameter *mtxid* (existence of locked mutexes, waiting tasks, etc.) into the area specified by parameter *pk_rmtx*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      mtxid = ID_MTX1;        /*Declares and initializes variable*/
    T_RMTX  pk_rmtx;                /*Declares data structure*/
    ID      htskid;                 /*Declares variable*/
    ID      wtskid;                 /*Declares variable*/
    ATR     mtxatr;                 /*Declares variable*/

    .........
    .........

    ref_mtx (mbxid, &pk_rmtx);      /*Reference mutex state*/

    htskid = pk_rmtx.htskid;        /*Acquires existence of locked mutexes*/
    wtskid = pk_rmtx.wtskid;        /*Reference ID number of the task at the */
                                    /*head of the wait queue*/
    mtxatr = pk_rmtx.mtxatr;        /*Reference attribute*/

    .........
    .........
}
```

Note For details about the mutex state packet, refer to "15.2.8 Mutex state packet".

CHAPTER 7 MEMORY POOL MANAGEMENT FUNCTIONS

This chapter describes the memory pool management functions performed by the RI850V4.

7.1 Outline

The statically secured memory areas in the Kernel Initialization Module are subject to management by the memory pool management functions of the RI850V4.

The RI850V4 provides a function to reference the memory area status, including the detailed information of fixed/variable-size memory pools, as well as a function to dynamically manipulate the memory area, including acquisition/release of fixed/variable-size memory blocks, by releasing a part of the memory area statically secured/initialized as "Fixed-Sized Memory Pools", or "Variable-Sized Memory Pools".

Table 7-1 Memory Area

セクション名	概要
.kernel_system	Area where executable code of RI850V4 is allocated.
.kernel_const	Area where static data of RI850V4 is allocated.
.kernel_data	Area where dynamic data of RI850V4 is allocated.
.kernel_data_init	Area where kernel initialization flag of RI850V4 is allocated.
.kernel_const_trace.const	Area where static data of trace function is allocated.
.kernel_data_trace.bss	Area where dynamic data of trace function is allocated.
.kernel_work	Area where system stack, task stack, data queue, fixed-sized memory pool and variable-sized memory pool is allocated.
.sec_nam（user-defined area）	Area where task stack, data queue, fixed-sized memory pool and variable-sized memory pool is allocated.

7.2 User-Own Coding Module

To support various execution environments, the hardware-dependent processing (Post-overflow processing) that is required for the RI850V4 to execute processing is extracted as a user-own coding module.

This enhances portability to various execution environments and facilitates customization as well.

Note The RI850V4 checks stack overflows only when "TA_ON: Overflow is checked" is defined as "Whether to check stack: stkchk" in the system configuration file.

7.2.1 Post-overflow processing

This is a routine dedicated to post-processing that is extracted as a user-own coding module to execute post-overflow processing and is called when a stack overflow occurs in a processing program.

- Basic form of post-overflow processing
  When coding the post-overflow processing, use a void function (function name: _kernel_stk_overflow) with two INT-type arguments.
  The "value of stack pointer sp when a stack overflow is detected" is set for the *r6* argument, and the "value of program counter pc when a stack overflow is detected" is set for the *r7* argument.
  The following shows the basic form of the post-overflow processing in assembly language.

```
#include    <kernel.h>              /*Standard header file definition*/

        .text
        .align  0x2
        .globl  __kernel_stk_overflow

__kernel_stk_overflow :

    .........

    .........

.halt_loop :
        jbr     .halt_loop
```

- Internal processing of post-overflow processing
  The overflow processing is a routine dedicated to post-processing that is extracted as a user-own coding module to execute post-overflow processing and is called when a stack necessary for the RI850V4 and the processing program has overflowed. Therefore, note the following points when coding post-overflow processing.

  - Coding method
    Code post-overflow processing using the C or assembly language.
    When coding in C, it can be coded in the same manner as ordinary functions.
    When coding in assembly language, code it according to the calling convention in the compiler used.
  - Stack switching
    The RI850V4 does not perform the processing related to stack switching when passing control to post-overflow processing. Therefore, when using the stack for post-overflow processing, the code for stack setting (setting of the stack pointer SP) should be written at the beginning of post-overflow processing.
  - Service call issue
    Issue of service calls is prohibited in post-overflow processing because correct operation cannot be guaranteed.

  The following is a list of processes that should be executed in post-overflow processing.

  - Post-processing that handles stack overflows

  Note The processing (such as reset) that should be coded as post-overflow processing depends on the user system.

7.3 Fixed-Sized Memory Pools

When a dynamic memory manipulation request is issued from a processing program in the RI850V4, the fixed-sized memory pool is provided as a usable memory area.

Dynamic memory manipulation of the fixed-size memory pool is executed in fixed size memory block units.

7.3.1 Create fixed-sized memory pool

In the RI850V4, the method of creating a fixed-sized memory pool is limited to "static creation".

Fixed-sized memory pools therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static fixed-size memory pool creation means defining of fixed-size memory pools using static API "CRE_MPF" in the system configuration file.

For details about the static API "CRE_MPF", refer to "17.5.7 Fixed-sized memory pool information".

7.3.2 Acquire fixed-sized memory block

A fixed-sized memory block is acquired (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- get_mpf
  This service call acquires the fixed-sized memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.
  If no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, this service call does not acquire the fixed-size memory block but queues the invoking task to the target fixed-size memory pool wait queue and moves it from the RUNNING state to the WAITING state (fixed-size memory block acquisition wait state).
  The WAITING state for a fixed-sized memory block is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Fixed-sized Memory Block Cancel Operation	Return Value
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf.	E_OK
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing irel_mpf.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mpfid = 1;              /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/

    .........
    .........

    ercd = get_mpf (mpfid, &p_blk); /*Acquire fixed-sized memory block */
                                    /*(waiting forever)*/

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........

        rel_mpf (mpfid, p_blk);     /*Release fixed-sized memory block*/
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }

        .........
        .........
}
```

Note 1 The RI850V4 does not perform memory clear processing when getting the acquired fixed-size memory block. The contents of the got fixed-size memory block are therefore undefined.

Note 2 Invoking tasks are queued to the target fixed-size memory pool wait queue in the order defined during configuration (FIFO order or priority order).

Note 3 If the fixed-size memory block acquisition wait state is cancelled because rel_wai or irel_wai was issued, the contents in the area specified by parameter *p_blk* become undefined.

- pget_mpf, ipget_mpf

This service call acquires the fixed-sized memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.
If a fixed-sized memory block could not be acquired from the target fixed-sized memory pool (no available fixed-sized memory blocks exist) when this service call is issued, fixed-sized memory block acquisition processing is not performed but "E_TMOUT" is returned.
The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mpfid = 1;              /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/

    .........
    .........

                                    /*Acquire fixed-sized memory block (polling)*/
    ercd = pget_mpf (mpfid, &p_blk);

    if (ercd == E_OK) {
        .........                   /*Polling success processing*/
        .........

        rel_mpf (mpfid, p_blk);     /*Release fixed-sized memory block*/
    } else if (ercd == E_TMOUT) {
        .........                   /*Polling failure processing*/
        .........
    }

    .........
    .........
}
```

Note 1 The RI850V4 does not perform memory clear processing when getting the acquired fixed-size memory block. The contents of the got fixed-size memory block are therefore undefined.

Note 2 If no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, the contents in the area specified by parameter *p_blk* become undefined.

- tget_mpf

This service call acquires the fixed-sized memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.
If no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, this service call does not acquire the fixed-size memory block but queues the invoking task to the target fixed-size memory pool wait queue and moves it from the RUNNING state to the WAITING state with timeout (fixed-size memory block acquisition wait state).
The WAITING state for a fixed-sized memory block is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Fixed-sized Memory Block Cancel Operation	Return Value
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf.	E_OK
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing irel_mpf.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mpfid = 1;              /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

    .........
    .........
                                    /*Acquire fixed-sized memory block*/
                                    /*(with timeout)*/
    ercd = tget_mpf (mpfid, &p_blk, tmout);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........

        rel_mpf (mpfid, p_blk);     /*Release fixed-sized memory block*/
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Timeout processing*/
        .........
    }

    .........
    .........
}
```

Note 1 The RI850V4 does not perform memory clear processing when getting the acquired fixed-size memory block. The contents of the got fixed-size memory block are therefore undefined.

Note 2 Invoking tasks are queued to the target fixed-size memory pool wait queue in the order defined during configuration (FIFO order or priority order).

Note 3 If the fixed-size memory block acquisition wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_blk* become undefined.

Note 4 TMO_FEVR is specified for wait time *tmout*, processing equivalent to get_mpf will be executed. When TMO_POL is specified, processing equivalent to pget_mpf /ipget_mpf will be executed.

7.3.3 Release fixed-sized memory block

A fixed-sized memory block is returned by issuing the following service call from the processing program.

- rel_mpf, irel_mpf
  This service call returns the fixed-sized memory block specified by parameter *blk* to the fixed-sized memory pool specified by parameter *mpfid*.
  If a task is queued to the target fixed-sized memory pool wait queue when this service call is issued, fixed-sized memory block return processing is not performed but fixed-sized memory blocks are returned to the relevant task (first task of wait queue).
  As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (WAITING state for a fixed-sized memory block) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mpfid = 1;              /*Declares and initializes variable*/
    VP      blk;                    /*Declares variable*/

    .........
    .........

    ercd = get_mpf (mpfid, &blk);   /*Acquire fixed-sized memory block */
                                    /*(waiting forever)*/

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........

        rel_mpf (mpfid, blk);       /*Release fixed-sized memory block*/
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

Note 1 The RI850V4 does not perform memory clear processing when returning the acquired fixed-size memory block. The contents of the returned fixed-size memory block are therefore undefined.

Note 2 When returning fixed-size memory blocks, be sure to issue either of these service calls for the acquired fixed-size memory pools. If the service call is issued for another fixed-size memory pool, no error results but the operation is not guaranteed after that.

7.3.4 Reference fixed-sized memory pool state

A fixed-sized memory pool status is referenced by issuing the following service call from the processing program.

- ref_mpf, iref_mpf
  Stores fixed-sized memory pool state packet (ID number of the task at the head of the wait queue, number of free memory blocks, etc.) of the fixed-sized memory pool specified by parameter *mpfid* in the area specified by parameter *pk_rmpf*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      mpfid = 1;              /*Declares and initializes variable*/
    T_RMPF  pk_rmpf;                /*Declares data structure*/
    ID      wtskid;                 /*Declares variable*/
    UINT    fblkcnt;                /*Declares variable*/
    ATR     mpfatr;                 /*Declares variable*/

    .........
    .........

    ref_mpf (mpfid, &pk_rmpf);      /*Reference fixed-sized memory pool state*/

    wtskid = pk_rmpf.wtskid;        /*Reference ID number of the task at the */
                                    /*head of the wait queue*/
    fblkcnt = pk_rmpf.fblkcnt;      /*Reference number of free memory blocks*/
    mpfatr = pk_rmpf.mpfatr;        /*Reference attribute*/

    .........
    .........
}
```

Note For details about the fixed-sized memory pool state packet, refer to "15.2.9 Fixed-sized memory pool state packet".

7.4 Variable-Sized Memory Pools

When a dynamic memory manipulation request is issued from a processing program in the RI850V4, the variable-sized memory pool is provided as a usable memory area.

Dynamic memory manipulation for variable-size memory pools is performed in the units of the specified variable-size memory block size.

7.4.1 Create variable-sized memory pool

In the RI850V4, the method of creating a variable-sized memory pool is limited to "static creation".

Variable-sized memory pools therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static variable-size memory pool creation means defining of variable-size memory pools using static API "CRE_MPL" in the system configuration file.

For details about the static API "CRE_MPL", refer to "17.5.8 Variable-sized memory pool information".

7.4.2 Acquire variable-sized memory block

A variable-sized memory block is acquired (waiting forever, polling, or with timeout) by issuing the following service call from the processing program.

- get_mpl
  This service call acquires a variable-size memory block of the size (+4 byte) specified by parameter *blksz* from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.
  If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory blocks but queues the invoking task to the target variable-size memory pool wait queue and moves it from the RUNNING state to the WAITING state (variable-size memory block acquisition wait state).
  The WAITING state for a variable-sized memory block is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Variable-sized Memory Block Cancel Operation	Return Value
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing rel_mpl.	E_OK
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing irel_mpl.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mplid = ID_MPL1;        /*Declares and initializes variable*/
    UINT    blksz = 256;            /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/

    .........
    .........
                                    /*Acquire variable-sized memory block */
                                    /*(waiting forever)*/
    ercd = get_mpl (mplid, blksz, &p_blk);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........

        rel_mpl (mplid, p_blk);     /*Release variable-sized memory block*/
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

Note 1 The RI850V4 acquires variable-size memory blocks in the unit of "integral multiple of 4". If a value other than an integral multiple of 4 is specified for parameter *blksz*, it is rounded up to be an integral multiple of 4.

Note 2 The RI850V4 needs a 4-byte area (management block) to manage the acquired variable-sized memory blocks. When this service call is issued, an area of "*blksz* + 4" bytes is allocated in the target variable-sized memory pool.

Note 3 The RI850V4 does not perform memory clear processing when getting the acquired variable-size memory block. The contents of the got variable-size memory block are therefore undefined.

Note 4 Invoking tasks are queued to the target variable-size memory pool wait queue in the order defined during configuration (FIFO order or priority order).

Note 5 If the variable-size memory block acquisition wait state is cancelled because rel_wai or irel_wai was issued, the contents in the area specified by parameter *p_blk* become undefined.

- pget_mpl, ipget_mpl

  This service call acquires a variable-size memory block of the size (+4 byte) specified by parameter *blksz* from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.

  If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory block but returns E_TMOUT.

  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mplid = ID_MTX1;        /*Declares and initializes variable*/
    UINT    blksz = 256;            /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/

    .........
    .........

                                    /*Acquire variable-sized memory block*/
                                    /*(polling)*/
    ercd = pget_mpl (mplid, blksz, &p_blk);

    if (ercd == E_OK) {
        .........                   /*Polling success processing*/
        .........

        rel_mpl (mplid, p_blk);     /*Release variable-sized memory block*/
    } else if (ercd == E_TMOUT) {
        .........                   /*Polling failure processing*/
        .........
    }

    .........
    .........
}
```

Note 1 The RI850V4 acquires variable-size memory blocks in the unit of "integral multiple of 4". If a value other than an integral multiple of 4 is specified for parameter *blksz*, it is rounded up to be an integral multiple of 4.

Note 2 The RI850V4 needs a 4-byte area (management block) to manage the acquired variable-sized memory blocks. When this service call is issued, an area of "*blksz* + 4" bytes is allocated in the target variable-sized memory pool.

Note 3 The RI850V4 does not perform memory clear processing when getting the acquired variable-size memory block. The contents of the got variable-size memory block are therefore undefined.

Note 4 If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, the contents in the area specified by parameter *p_blk* become undefined.

- tget_mpl
  This service call acquires a variable-size memory block of the size (+4 byte) specified by parameter *blksz* from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.
  If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory blocks but queues the invoking task to the target variable-size memory pool wait queue and moves it from the RUNNING state to the WAITING state with timeout (variable-size memory block acquisition wait state).
  The WAITING state for a variable-sized memory block is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Variable-sized Memory Block Cancel Operation	Return Value
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing rel_mpl.	E_OK
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing irel_mpl.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void
task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mplid = ID_MPL1;        /*Declares and initializes variable*/
    UINT    blksz = 256;            /*Declares and initializes variable*/
    VP      p_blk;                  /*Declares variable*/
    TMO     tmout = 3600;           /*Declares and initializes variable*/

    .........
    .........

                                    /*Acquire variable-sized memory block*/
                                    /*(with timeout)*/
    ercd = tget_mpl (mplid, blksz, &p_blk, tmout);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........

        rel_mpl (mplid, p_blk ;     /*Release variable-sized memory block*/
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    } else if (ercd == E_TMOUT) {
        .........                   /*Timeout processing*/
        .........
    }

    .........
    .........
}
```

Note 1 The RI850V4 acquires variable-size memory blocks in the unit of "integral multiple of 4". If a value other than an integral multiple of 4 is specified for parameter *blksz*, it is rounded up to be an integral multiple of 4.

Note 2 The RI850V4 needs a 4-byte area (management block) to manage the acquired variable-sized memory blocks. When this service call is issued, an area of "*blksz* + 4" bytes is allocated in the target variable-sized memory pool.

Note 3 The RI850V4 does not perform memory clear processing when getting the acquired variable-size memory block. The contents of the got variable-size memory block are therefore undefined.

Note 4 Invoking tasks are queued to the target variable-size memory pool wait queue in the order defined during configuration (FIFO order or priority order).

Note 5 If the variable-size memory block acquisition wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_blk* become undefined.

Note 6 TMO_FEVR is specified for wait time *tmout*, processing equivalent to get_mpl will be executed. When TMO_POL is specified, processing equivalent to pget_mpl /ipget_mpl will be executed.

7.4.3 Release variable-sized memory block

A variable-sized memory block is returned by issuing the following service call from the processing program.

- rel_mpl, irel_mpl
  This service call returns the variable-sized memory block specified by parameter *blk* to the variable-sized memory pool specified by parameter *mplid*.
  After returning the variable-size memory blocks, these service calls check the tasks queued to the target variable-size memory pool wait queue from the top, and assigns the memory if the size of memory requested by the wait queue is available. This operation continues until no tasks queued to the wait queue remain or no memory space is available. As a result, the task that acquired the memory is unlinked from the queue and moved from the WAITING state (variable-size memory block acquisition wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER      ercd;                   /*Declares variable*/
    ID      mplid = ID_MPL1;              /*Declares and initializes variable*/
    UINT    blksz = 256;            /*Declares and initializes variable*/
    VP      blk;                    /*Declares variable*/

    .........
    .........

                                    /*Acquire variable-sized memory block*/
    ercd = get_mpl (mplid, blksz, &blk);

    if (ercd == E_OK) {
        .........                   /*Normal termination processing*/
        .........

        rel_mpl (mplid, blk);       /*Release variable-sized memory block*/
    } else if (ercd == E_RLWAI) {
        .........                   /*Forced termination processing*/
        .........
    }

    .........
    .........
}
```

Note 1 The RI850V4 does not perform memory clear processing when returning the acquired variable-size memory block. The contents of the returned variable-size memory block are therefore undefined.

Note 2 When returning variable-size memory blocks, be sure to issue either of these service calls for the acquired variable-size memory pools. If the service call is issued for another variable-size memory pool, no error results but the operation is not guaranteed after that.

7.4.4 Reference variable-sized memory pool state

A variable-sized memory pool status is referenced by issuing the following service call from the processing program.

- ref_mpl, iref_mpl
  These service calls store the detailed information (ID number of the task at the head of the wait queue, total size of free memory blocks, etc.) of the variable-size memory pool specified by parameter *mplid* into the area specified by parameter *pk_rmpl*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      mplid = ID_MPL1;        /*Declares and initializes variable*/
    T_RMPL  pk_rmpl;                /*Declares data structure*/
    ID      wtskid;                 /*Declares variable*/
    SIZE    fmplsz;                 /*Declares variable*/
    UINT    fblksz;                 /*Declares variable*/
    ATR     mplatr;                 /*Declares variable*/

    .........
    .........

    ref_mpl (mplid, &pk_rmpl);      /*Reference variable-sized memory pool state*/

    wtskid = pk_rmpl.wtskid;        /*Reference ID number of the task at the */
                                    /*head of the wait queue*/
    fmplsz = pk_rmpl.fmplsz;        /*Reference total size of free memory blocks*/
    fblksz = pk_rmpl.fblksz;        /*Reference maximum memory block size*/
    mplatr = pk_rmpl.mplatr;        /*Reference attribute*/

    .........
    .........
}
```

Note For details about the variable-sized memory pool state packet, refer to "15.2.10 Variable-sized memory pool state packet".

CHAPTER 8 SYSTEM STATE MANAGEMENT FUNCTIONS

This chapter describes the system management functions performed by the RI850V4.

8.1 Outline

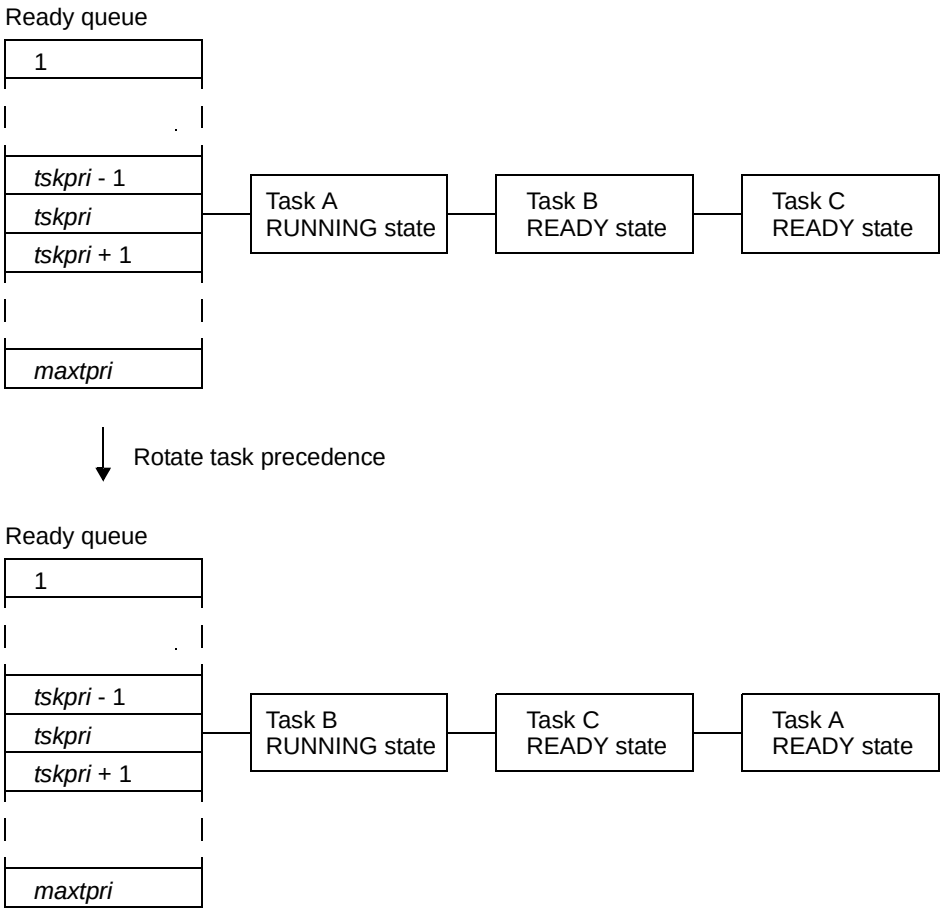
The RI850V4's system status management function provides functions for referencing the system status such as the context type and CPU lock status, as well as functions for manipulating the system status such as ready queue rotation, scheduler activation, or the like.

8.2 Rotate Task Precedence

A ready queue is rotated by issuing the following service call from the processing program.

- rot_rdq, irot_rdq
  This service call re-queues the first task of the ready queue corresponding to the priority specified by parameter *tskpri* to the end of the queue to change the task execution order explicitly.
  The following shows the status transition when this service call is used.

Figure 8-1 Rotate Task Precedence



The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void cychdr (VP_INT exinf)
{
    PRI     tskpri = 8;             /*Declares and initializes variable*/

    .........
    .........

    irot_rdq (tskpri);              /*Rotate task precedence*/

    .........
    .........

    return;                         /*Terminate cyclic handler*/
}
```

Note 1 This service call does not perform queuing of rotation requests. If no task is queued to the ready queue corresponding to the relevant priority, therefore, no processing is performed but it is not handled as an error.

Note 2 Round-robin scheduling can be implemented by issuing this service call via a cyclic handler in a constant cycle.

Note 3 The ready queue is a hash table that uses priority as the key, and tasks that have entered an executable state (READY state or RUNNING state) are queued in FIFO order.
Therefore, the scheduler realizes the RI850V4's scheduling system (priority level method, FCFS method) by executing task detection processing from the highest priority level of the ready queue upon activation, and upon detection of queued tasks, giving the CPU use right to the first task of the proper priority level.

8.3 Forced Scheduler Activation

The scheduler can be forcibly activated by issuing the following service call from the processing program.

- vsta_sch
  This service call explicitly forces the RI850V4 scheduler to activate. If a scheduling request has been kept pending, task switching may therefore occur.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{

    .........
    .........

    vsta_sch ();                    /*Forced scheduler*/

    .........
    .........
}
```

Note The RI850V4 provides this service call as a function to activate a scheduler from a task for which preempt acknowledge status (TA_DISPREEMPT) disable is defined during configuration.

8.4 Reference Task ID in the RUNNING State

A RUNNING-state task is referenced by issuing the following service call from the processing program.

- get_tid, iget_tid
  These service calls store the ID of a task in the RUNNING state in the area specified by parameter *p_tskid*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void inthdr (void)
{
    ID      p_tskid;                /*Declares variable*/

    .........
    .........

    iget_tid (&p_tskid);            /*Reference task ID in the RUNNING state*/

    .........
    .........

    return;                         /*Terminate interrupt handler*/
}
```

Note This service call stores TSK_NONE in the area specified by parameter *p_tskid* if no tasks that have entered the RUNNING state exist (all tasks in the IDLE state).

8.5 Lock the CPU

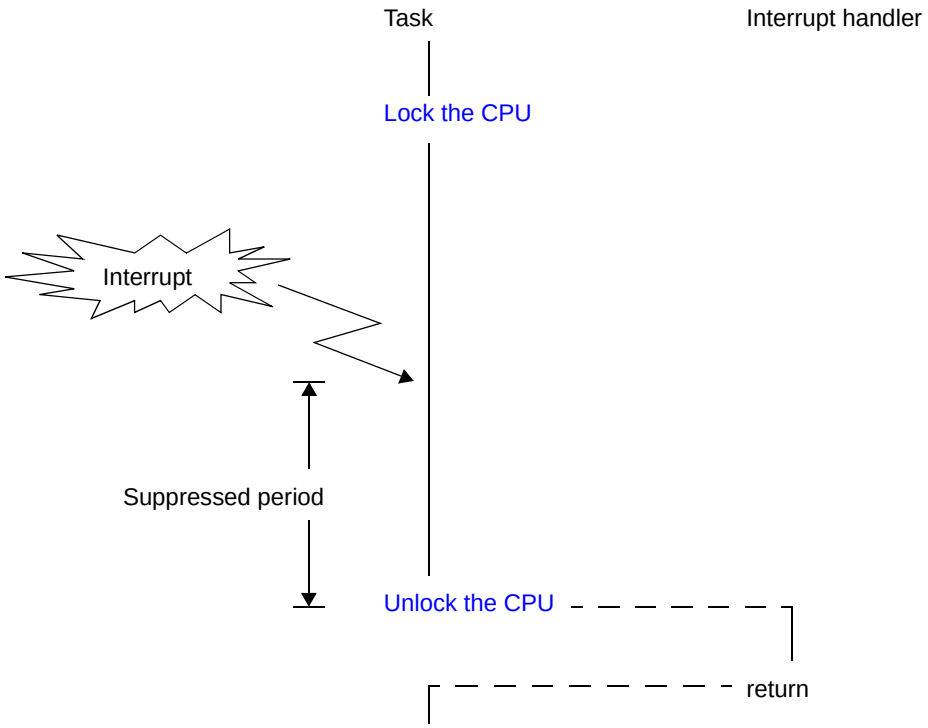
A task is moved to the CPU locked state by issuing the following service call from the processing program.

- loc_cpu, iloc_cpu
  These service calls change the system status type to the CPU locked state.
  As a result, EI level maskable interrupt acknowledgment processing is prohibited during the interval from this service call is issued until unl_cpu or iunl_cpu is issued, and service call issue is also restricted.
  The service calls that can be issued in the CPU locked state are limited to the one listed below.

Service Call	Function
loc_cpu, iloc_cpu	Lock the CPU.
unl_cpu, iunl_cpu	Unlock the CPU.
sns_loc	Reference CPU state.
sns_dsp	Reference dispatching state.
sns_ctx	Reference contexts.
sns_dpn	Reference dispatch pending state.

If an EI level maskable interrupt is created during this period, the RI850V4 delays transition to the relevant interrupt processing (interrupt handler) until either unl_cpu or iunl_cpu is issued.
The following shows a processing flow when using this service call.

Figure 8-2 Lock the CPU



The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    .........
    .........

    loc_cpu ();                     /*Lock the CPU*/

    .........                       /*CPU locked state*/
    .........

    unl_cpu ();                     /*Unlock the CPU*/

    .........
    .........
}
```

Note 1 The CPU locked state changed by issuing this service call must be cancelled before the processing program that issued this service call ends.

Note 2 This service call does not perform queuing of lock requests. If the system is in the CPU locked state, therefore, no processing is performed but it is not handled as an error.

Note 3 This service call manipulates PM*n* bits in the priority mask register (PMR) to disable acceptance of EI level maskable interrupts.
The PM*n* bits to be manipulated correspond to the interrupt priority range defined as the Maximum interrupt priority: maxintpri during configuration.

Note 4 This service call does not manipulate the ID bit in the program status word (PSW).

Note 5 The RI850V4 realizes the TIME MANAGEMENT FUNCTIONS by using base clock timer interrupts that occur at constant intervals. If acknowledgment of the relevant base clock timer interrupt is disabled by issuing this service call, the TIME MANAGEMENT FUNCTIONS may no longer operate normally.

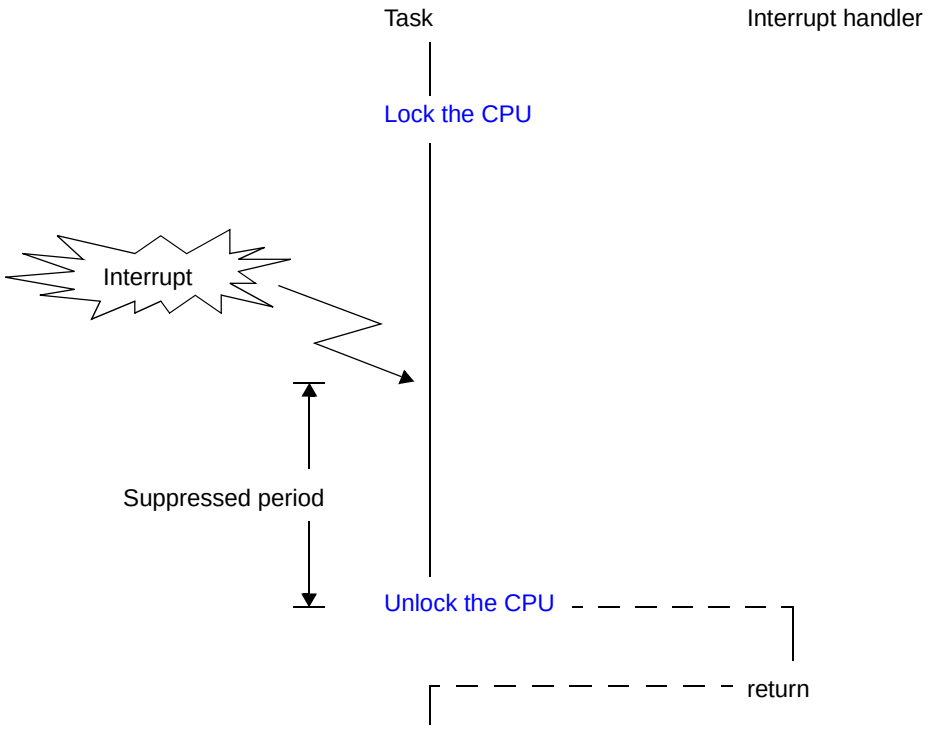
Note 6 If this service call or a service call other than sns_*xxx* is issued from when this service call is issued until unl_cpu or iunl_cpu is issued, the RI850V4 returns E_CTX.

8.6 Unlock the CPU

The CPU locked state is cancelled by issuing the following service call from the processing program.

- unl_cpu, iunl_cpu
  These service calls change the system status to the CPU unlocked state.
  As a result, acknowledge processing of EI level maskable interrupts prohibited through issue of either loc_cpu or iloc_cpu is enabled, and the restriction on service call issue is released.
  If an EI level maskable interrupt is created during the interval from when either loc_cpu or iloc_cpu is issued until this service call is issued, the RI850V4 delays transition to the relevant interrupt processing (interrupt handler) until this service call is issued.
  The following shows a processing flow when using this service call.

Figure 8-3 Unlock the CPU



The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    .........
    .........

    loc_cpu ();                     /*Lock the CPU*/

    .........                       /*CPU locked state*/
    .........

    unl_cpu ();                     /*Unlock the CPU*/
```

```
    .........
    .........
}
```

Note 1 This service call does not perform queuing of cancellation requests. If the system is in the CPU unlocked state, therefore, no processing is performed but it is not handled as an error.

Note 2 This service call manipulates PM*n* bits in the priority mask register (PMR) to disable acceptance of EI level maskable interrupts.
The PM*n* bits to be manipulated correspond to the interrupt priority range defined as the Maximum interrupt priority: maxintpri during configuration.

Note 3 This service call does not cancel the dispatch disabled state that was set by issuing dis_dsp. If the system status before the CPU locked state is entered was the dispatch disabled state, the system status becomes the dispatch disabled state after this service call is issued.

Note 4 If a service call other than loc_cpu, iloc_cpu and sns_*xxx* is issued from when loc_cpu or iloc_cpu is issued until this service call is issued, the RI850V4 returns E_CTX.

8.7 Reference CPU State

The CPU locked state is referenced by issuing the following service call from the processing program.

- sns_loc
  This service call acquires the system status type when this service call is issued (CPU locked state or CPU unlocked state).
  When this service call is terminated normally, the acquired system state type (TRUE: CPU locked state, FALSE: CPU unlocked state) is returned.
  The following describes an example for coding this service call.

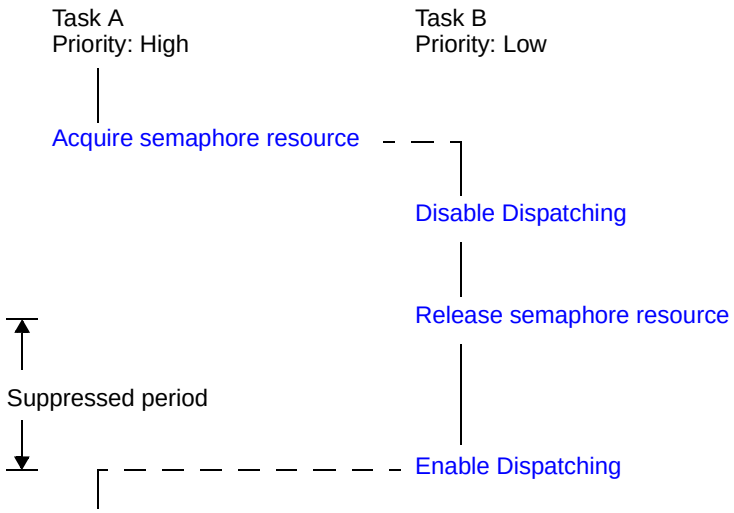
```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    BOOL ercd;                      /*Declares variable*/

    .........
    .........

    ercd = sns_loc ();              /*Reference CPU state*/

    if (ercd == TRUE) {
        .........                   /*CPU locked state*/
        .........
    } else if (ercd == FALSE) {
        .........                   /*CPU unlocked state*/
        .........
    }

    .........
    .........
}
```

8.8 Disable Dispatching

A task is moved to the dispatch disabled state by issuing the following service call from the processing program.

- dis_dsp
  This service call changes the system status to the dispatch disabled state.
  As a result, dispatch processing (task scheduling) is disabled from when this service call is issued until ena_dsp is issued.
  If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when this service call is issued until ena_dsp is issued, the RI850V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until ena_dsp is issued, upon which the actual dispatch processing is performed in batch.
  The following shows a processing flow when using this service call.

Figure 8-4 Disable Dispatching



The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

#pragma rtos_task   task            /*#pragma directive definition*/

void task (VP_INT exinf)
{
    .........
    .........

    dis_dsp ();                     /*Disable dispatching*/

    .........                       /*Dispatching disabled state*/
    .........

    ena_dsp ();                     /*Enable dispatching*/

    .........
    .........
}
```

Note 1 The dispatch disabled state changed by issuing this service call must be cancelled before the task that issued this service call moves to the DORMANT state.

Note 2 This service call does not perform queuing of disable requests. If the system is in the dispatch disabled state, therefore, no processing is performed but it is not handled as an error.

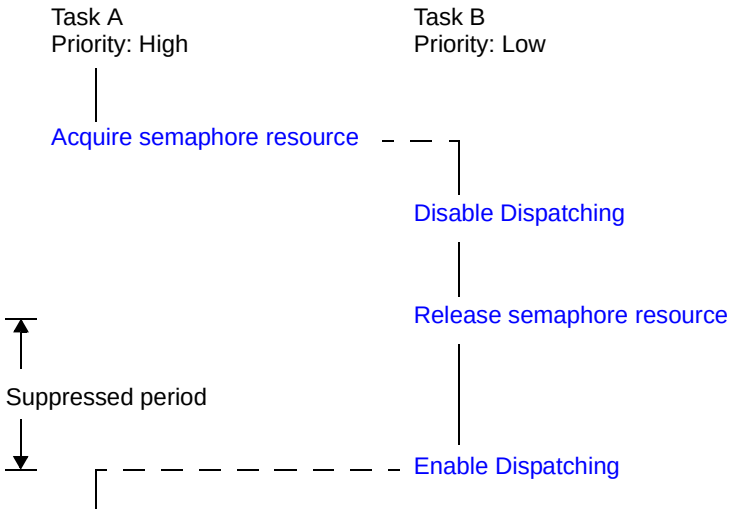
Note 3 If a service call (such as wai_sem, wai_flg) that may move the status of an invoking task is issued from when this service call is issued until ena_dsp is issued, the RI850V4 returns E_CTX regardless of whether the required condition is immediately satisfied.

8.9 Enable Dispatching

The dispatch disabled state is cancelled by issuing the following service call from the processing program.

- ena_dsp
  This service call changes the system status to the dispatch enabled state.
  As a result, dispatch processing (task scheduling) that has been disabled by issuing dis_dsp is enabled.
  If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when dis_dsp is issued until this service call is issued, the RI850V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until this service call is issued, upon which the actual dispatch processing is performed in batch.
  The following shows a processing flow when using this service call.

Figure 8-5 Enable Dispatching



The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    .........
    .........

    dis_dsp ();                     /*Disable dispatching*/

    .........                       /*Dispatching disabled state*/
    .........

    ena_dsp ();                     /*Enable dispatching*/

    .........
    .........
}
```

Note 1 This service call does not perform queuing of enable requests. If the system is in the dispatch enabled state, therefore, no processing is performed but it is not handled as an error.

Note 2 If a service call (such as wai_sem, wai_flg) that may move the status of an invoking task is issued from when dis_dsp is issued until this service call is issued, the RI850V4 returns E_CTX regardless of whether the required condition is immediately satisfied.

8.10 Reference Dispatching State

The dispatch disabled state is referenced by issuing the following service call from the processing program.

- sns_dsp
  This service call acquires the system status type when this service call is issued (dispatch disabled state or dispatch enabled state).
  When this service call is terminated normally, the acquired system state type (TRUE: dispatch disabled state, FALSE: dispatch enabled state) is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    BOOL ercd;                      /*Declares variable*/

    .........
    .........

    ercd = sns_dsp ();              /*Reference dispatching state*/

    if (ercd == TRUE) {
        .........                   /*Dispatching disabled state*/
        .........
    } else if (ercd == FALSE) {
        .........                  /*Dispatching enabled state*/
        .........
    }

    .........
    .........
}
```

8.11 Reference Contexts

The context type is referenced by issuing the following service call from the processing program.

- sns_ctx
  This service call acquires the context type of the processing program that issued this service call (non-task context or task context).
  When this service call is terminated normally, the acquired context type (TRUE: non-task context, FALSE: task context) is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    BOOL ercd;                      /*Declares variable*/

    .........
    .........

    ercd = sns_ctx ();              /*Reference contexts*/

    if (ercd == TRUE) {
        .........                   /*Non-task contexts*/
        .........
    } else if (ercd == FALSE) {
        .........                   /*Task contexts*/
        .........
    }

    .........
    .........
}
```

8.12 Reference Dispatch Pending State

The dispatch pending state is referenced by issuing the following service call from the processing program.

- sns_dpn
  This service call acquires the system status type when this service call is issued (whether in dispatch pending state or not).
  When this service call is terminated normally, the acquired system state type (TRUE: dispatch pending state, FALSE: dispatch not-pending state) is returned.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    BOOL ercd;                      /*Declares variable*/

    .........
    .........

    ercd = sns_dpn ();              /*Reference dispatch pending state*/

    if (ercd == TRUE) {
        .........                   /*Dispatch pending state*/
        .........
    } else if (ercd == FALSE) {
        .........                   /*Other state*/
        .........
    }

    .........
    .........
}
```

CHAPTER 9 TIME MANAGEMENT FUNCTIONS

This chapter describes the time management functions performed by the RI850V4.

9.1 Outline

The RI850V4's time management function provides methods to implement time-related processing (Timer Operations: Delayed task wakeup, Timeout, Cyclic handlers) by using base clock timer interrupts that occur at constant intervals, as well as a function to manipulate and reference the system time.

9.2 System Time

The system time is a time used by the RI850V4 for performing time management (unit: millisecond).

After initialization by the Kernel Initialization Module, the system time is updated based on the Base clock interval: tim_base when an EI level maskable interrupt defined in the Base clock timer exception code: tim_intno in the system configuration file occurs.

9.2.1 Base clock timer interrupt

To realize the time management function, the RI850V4 uses interrupts that occur at constant intervals (base clock timer interrupts).

When a base clock timer interrupt occurs, processing related to the RI850V4 time (system time update, task timeout/ delay, cyclic handler activation, etc.) is executed.

A base clock timer interrupt is caused by an EI level maskable interrupt defined in the Base clock timer exception code: tim_intno in the system configuration file.

For details about the basic information "CLK_INTNO", refer to "17.4.2 Basic information".

The RI850V4 does not initialize hardware to generate base clock timer interrupts, so it must be coded by the user.

Initialize the hardware used by Boot processing or Initialization routine and cancel the interrupt masking.

The following shows the necessary settings when using the OS timer as the base clock timer.

OS Timer Setting Registers	Necessary Setting
OSTMn control register (OSTMnCTL)	OSTMnCTL.OSTMnMD1 = 0
OSTMn compare register (OSTMnCMP)	OSTMnCMP=(TIC_NUME*1000000) / KERNEL_USR_BASETIME
Timer interrupt priority	Maximum interrupt priority: maxintpri or a lower value

Note When passing control to the processing related to the base clock timer interrupt, the RI850V4 enables acceptance of EI level maskable interrupts by manipulating the PM*n* bits in the priority mask register (PMR) and the ID bit in the program status word (PSW), and issuing the eiret instruction (clearing the in-service priority register (ISPR)).
Therefore, if an EI level maskable interrupt occurs within the base clock timer interrupt processing, the interrupt is accepted.

Note Use the OS timer in the interval timer mode.

9.2.2 Base clock interval

In the RI850V4, service call parameters for time specification are specified in milliseconds.

If is desirable to set 1 ms for the occurrence interval of base clock timer interrupts, but it may be difficult depending on the target system performance (processing capability, required time resolution, or the like).

The interval between occurrences of base clock timer interrupts can be defined as the Base clock interval: tim_base in the system configuration file.

By specifying the base clock cycle, processing regards that the time equivalent to the base clock cycle elapses during a base clock timer interrupt.

An integer value larger than 1 can be specified for the base clock cycle. Floating-point values such as 2.5 cannot be specified.

9.3 Timer Operations

The RI850V4's timer operation function provides Delayed task wakeup, Timeout and Cyclic handlers, as the method for realizing time-dependent processing.

9.3.1 Delayed task wakeup

Delayed wakeup the operation that makes the invoking task transit from the RUNNING state to the WAITING state during the interval until a given length of time has elapsed, and makes that task move from the WAITING state to the READY state once the given length of time has elapsed.

Delayed wakeup is implemented by issuing the following service call from the processing program.

dly_tsk

9.3.2 Timeout

Timeout is the operation that makes the target task move from the RUNNING state to the WAITING state during the interval until a given length of time has elapsed if the required condition issued from a task is not immediately satisfied, and makes that task move from the WAITING state to the READY state regardless of whether the required condition is satisfied once the given length of time has elapsed.

A timeout is implemented by issuing the following service call from the processing program.

tslp_tsk, twai_sem, twai_flg, tsnd_dtq, trcv_dtq, trcv_mbx, tloc_mtx, tget_mpf, tget_mpl

9.3.3 Cyclic handlers

The cyclic handler is a routine dedicated to cycle processing that is activated periodically at a constant interval (activation cycle).

The RI850V4 handles the cyclic handler as a "non-task (module independent from tasks)". Therefore, even if a task with the highest priority in the system is being executed, the processing is suspended when a specified activation cycle has come, and the control is passed to the cyclic handler.

The RI850V4 manages the states in which each cyclic handler may enter and cyclic handlers themselves, by using management objects (cyclic handler control blocks) corresponding to cyclic handlers one-to-one.

- Basic form of cyclic handlers
  When coding a cyclic handler, use a void function with one VP_INT argument (any function name is fine).
  The extended information specified with Cyclic handler information is set for the *exinf* argument.
  The following shows the basic form of cyclic handlers in C.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void cychdr (VP_INT exinf)
{
    .........
    .........

    return;                         /*Terminate cyclic handler*/
}
```

- Coding method
  Code cyclic handlers using C or assembly language.
  When coding in C, they can be coded in the same manner as void type functions coded.
  When coding in assembly language, code them according to the calling rules prescribed in the compiler used.

- Stack switching
  The RI850V4 switches to the system stack specified in the Basic information when passing control to a cyclic handler, and switches to the relevant stack when returning control from the cyclic handler to the processing program in which a base clock timer interrupt occurred and caused activation of the cyclic handler. Therefore, coding regarding stack switching is not required in a cyclic handler.

- Service call issue
  The RI850V4 handles the cyclic handler as a "non-task".
  Service calls that can be issued in cyclic handlers are limited to the service calls that can be issued from non-tasks.

  Note 1 If a service call (isig_sem, iset_flg, etc.) accompanying dispatch processing (task scheduling processing) is issued in order to quickly complete the processing in the cyclic handler during the interval until the processing in the cyclic handler ends, the RI850V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until a return instruction is issued by the cyclic handler, upon which the actual dispatch processing is performed in batch.

  Note 2 For details on the valid issue range of each service call, refer to Table 16-1 to Table 16-12.

- Acceptance of EI level maskable interrupts
  When passing control to a cyclic handler, the RI850V4 enables acceptance of EI level maskable interrupts by manipulating the PM*n* bits in the priority mask register (PMR) and the ID bit in the program status word (PSW), and issuing the eiret instruction (clearing the in-service priority register (ISPR)).
  Therefore, if an EI level maskable interrupt occurs within a cyclic handler, the interrupt is accepted.

9.3.4 Create cyclic handler

In the RI850V4, the method of creating a cyclic handler is limited to "static creation".

Cyclic handlers therefore cannot be created dynamically using a method such as issuing a service call from a processing program.

Static cyclic handler creation means defining of cyclic handlers using static API "CRE_CYC" in the system configuration file.

For details about the static API "CRE_CYC", refer to "17.5.9 Cyclic handler information".

9.4 Set System Time

The system time can be set by issuing the following service call from the processing program.

- set_tim, iset_tim
  These service calls change the RI850V4 system time (unit: millisecond) to the time specified by parameter *p_systim*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    SYSTIM  p_systim;               /*Declares data structure*/

    p_systim.ltime = 3600;          /*Initializes data structure*/
    p_systim.utime = 0;             /*Initializes data structure*/

    .........
    .........

    set_tim (&p_systim);            /*Set system time*/

    .........
    .........
}
```

Note For details about the system time packet, refer to "15.2.11 System time packet".

9.5 Reference System Time

The system time can be referenced by issuing the following service call from the processing program.

- get_tim, iget_tim
  These service calls store the RI850V4 system time (unit: millisecond) into the area specified by parameter *p_systim*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    SYSTIM  p_systim;               /*Declares data structure*/
    UW      ltime;                  /*Declares variable*/
    UH      utime;                  /*Declares variable*/

    .........
    .........

    get_tim (&p_systim);            /*Reference System Time*/

    ltime = p_systim.ltime;         /*Acquirer system time (lower 32 bits)*/
    utime = p_systim.utime;         /*Acquirer system time (higher 16 bits)*/

    .........
    .........
}
```

Note 1 The RI850V4 ignores the numeric values that cannot be expressed as the system time (values overflowed from the 48-bit width).

Note 2 For details about the system time packet, refer to "15.2.11 System time packet".

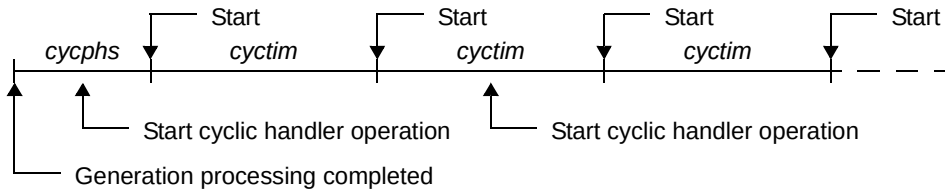
9.6 Start Cyclic Handler Operation

Moving to the operational state (STA state) is implemented by issuing the following service call from the processing program.

- sta_cyc, ista_cyc
  This service call moves the cyclic handler specified by parameter *cycid* from the non-operational state (STP state) to operational state (STA state).
  As a result, the target cyclic handler is handled as an activation target of the RI850V4.
  The relative interval from when either of this service call is issued until the first activation request is issued varies depending on whether the TA_PHS attribute is specified for the target cyclic handler during configuration.

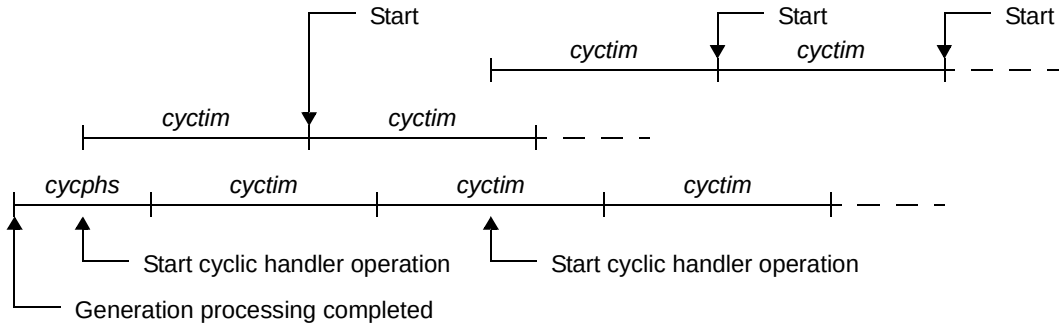
  - If the TA_PHS attribute is specified
    The target cyclic handler activation timing is set based on the activation phases (initial activation phase *cycphs* and activation cycle *cyctim*) defined during configuration.
    If the target cyclic handler has already been started, however, no processing is performed even if this service call is issued, but it is not handled as an error.
    The following shows a cyclic handler activation timing image.

Figure 9-1 TA_PHS Attribute: Specified



  - If the TA_PHS attribute is not specified
    The target cyclic handler activation timing is set based on the activation phase (activation cycle *cyctim*) when this service call is issued.
    This setting is performed regardless of the operating status of the target cyclic handler.
    The following shows a cyclic handler activation timing image.

Figure 9-2 TA_PHS Attribute: Not Specified



The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
```

```
{
    ID      cycid = ID_CYC1;        /*Declares and initializes variable*/

    .........
    .........

    sta_cyc (cycid);                /*Start cyclic handler operation*/

    .........
    .........
}
```

Note The extended information specified in the Cyclic handler information is passed to the cyclic handler activated by issuing this service call.

9.7 Stop Cyclic Handler Operation

Moving to the non-operational state (STP state) is implemented by issuing the following service call from the processing program.

- stp_cyc, istp_cyc
  This service call moves the cyclic handler specified by parameter *cycid* from the operational state (STA state) to non-operational state (STP state).
  As a result, the target cyclic handler is excluded from activation targets of the RI850V4 until issue of sta_cyc or ista_cyc.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      cycid = ID_CYC1;        /*Declares and initializes variable*/

    .........
    .........

    stp_cyc (cycid);                /*Stop cyclic handler operation*/

    .........
    .........
}
```

Note This service call does not perform queuing of stop requests. If this service call has already been issued and the target cyclic handler has been moved to the non-operational state (STP state), no processing is performed but it is not handled as an error.

9.8 Reference Cyclic Handler State

A cyclic handler status by issuing the following service call from the processing program.

- ref_cyc, iref_cyc
  Stores cyclic handler state packet (current state, time left before the next activation, etc.) of the cyclic handler specified by parameter *cycid* in the area specified by parameter *pk_rcyc*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ID      cycid = ID_CYC1;        /*Declares and initializes variable*/
    T_RCYC  pk_rcyc;                /*Declares data structure*/
    STAT    cycstat;                /*Declares variable*/
    RELTIM  lefttim;                /*Declares variable*/
    ATR     cycatr;                 /*Declares variable*/
    RELTIM  cyctim;                 /*Declares variable*/
    RELTIM  cycphs;                 /*Declares variable*/

    .........
    .........

    ref_cyc (cycid, &pk_rcyc);      /*Reference cyclic handler state*/

    cycstat = pk_rcyc.cycstat;      /*Reference current state*/
    lefttim = pk_rcyc.lefttim;      /*Reference time left before the next */
                                    /*activation*/
    cycatr = pk_rcyc.cycatr;        /*Reference attribute*/
    cyctim = pk_rcyc.cyctim;        /*Reference activation cycle*/
    cycphs = pk_rcyc.cycphs;        /*Reference activation phase*/

    .........
    .........
}
```

Note For details about the cyclic handler state packet, refer to "15.2.12 Cyclic handler state packet".

CHAPTER 10 INTERRUPT MANAGEMENT FUNCTIONS

This chapter describes the interrupt management functions performed by the RI850V4.

10.1 Outline

The RI850V4 provides as interrupt management functions related to the interrupt handlers activated when an EI level maskable interrupt is occurred.

10.2 User-Own Coding Module

To support various execution environments, the RI850V4 extracts from the interrupt management functions the hardware-dependent processing (Interrupt entry processing) that is required to execute processing, as a user-own coding module. This enhances portability for various execution environments and facilitates customization as well.

10.2.1 Interrupt entry processing

Interrupt entry processing is a routine dedicated to entry processing that is extracted as a user-own coding module to assign processing for branching to the relevant processing (such as interrupt preprocessing), to the handler address to which the CPU forcibly passes control when an interrupt occurs.

The interrupt entry processing for the EI level maskable interrupts defined in the Interrupt handler information in the system configuration file is included in the entry file created by executing the configurator for the system configuration file. Therefore, coding of interrupt entry processing is necessary for other interrupts (such as a reset) that are not EI level maskable interrupts.

- Basic form of interrupt entry processing
  When coding an interrupt entry processing, the code should match the branch method selected in the Property panel -> [System Configuration File Related Information] tabbed page -> [Entry File] category -> [Generate method].
  The following shows the basic form of interrupt entry processing in assembly language, related to other interrupts (such as EI level maskable interrupts not defined in the Interrupt handler information, reset, or FE level maskable interrupts) that are not EI level maskable interrupts defined in the Interrupt handler information in the system configuration file.
  For interrupt types other than the EI level maskable interrupt, interrupt entry processing should always be coded in the direct vector method.

  [Direct vector method]

```
    .extern     _inthdr                 -- External label declaration
    .org        base_adress + offset    -- Setting of branch destination address
    jr32        _inthdr                 -- Branch to interrupt processing
```

Note Set *base_address* to the same value as that specified in the Property panel -> [System Configuration File Related Information] tabbed page _ [Entry File] category -> [Specify an exception handler vector address].
Set *offset* to the offset value corresponding to the priority of the target interrupt.

[Table reference method]

```
    .extern     _inthdr                 -- External label declaration
    .org        base_adress + offset    -- Setting of branch destination address
    dw          !_inthdr                -- Branch to interrupt processing
```

Note Set *base_address* to the same value as that specified in the Property panel -> [System Configuration File Related Information] tabbed page _ [Entry File] category -> [Base address of the interrupt handler address table].
Set *offset* to the offset value corresponding to the source of the target interrupt.

- Internal processing of interrupt entry processing
Interrupt entry processing is a routine dedicated to entry processing that is called without RI850V4 intervention when an interrupt occurs. Therefore, note the following points when coding interrupt

  - Coding method
  Code it in assembly language according to the calling convention in the compiler used.

  - Stack switching
  There is no stack that requires switching before executing interrupt entry processing.
  Therefore, coding regarding stack switching is not required in interrupt entry processing.

  - Service call issue
  To achieve faster response for processing (such as an Interrupt Handlers) corresponding to an interrupt that has occurred, the issue of service calls is prohibited during interrupt entry processing.

The following is a list of processes that should be executed in interrupt entry processing.

  - External label declaration

  - Setting of branch destination address

  - Branch to interrupt processing

10.3 Interrupt Handlers

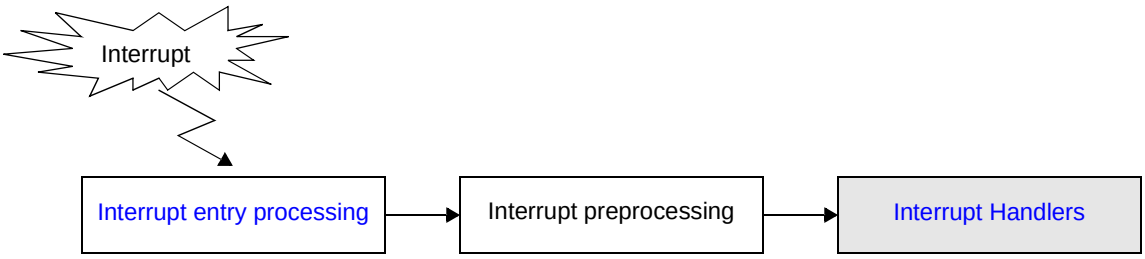
The interrupt handler is a routine dedicated to interrupt servicing that is activated when an EI level maskable interrupt occurs.

The RI850V4 handles the interrupt handler as a non-task (module independent from tasks). Therefore, even if a task with the highest priority in the system is being executed, the processing is suspended when an interrupt occurs, and the control is passed to the interrupt handler.

The RI850V4 manages the states in which each interrupt handler may enter and interrupt handlers themselves, by using management objects (interrupt handler control blocks) corresponding to interrupt handlers one-to-one.

The following shows a processing flow from when an interrupt occurs until the control is passed to the interrupt handler.

Figure 10-1 Processing Flow (Interrupt Handler)



10.3.1 Basic form of interrupt handlers

Code interrupt handlers by using the void type function that has no arguments.

The following shows the basic form of interrupt handlers in C.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void inthdr (void)
{
    .........
    .........

    return;                         /*Terminate interrupt handler*/
}
```

10.3.2 Internal processing of interrupt handler

The RI850V4 executes "original pre-interrupt processing" when passing control from the processing program where an EI level maskable interrupt occurred to the interrupt handler, as well as "original post-interrupt processing" when restoring control from the interrupt handler to the processing program where the EI level maskable interrupt occurred. Therefore, note the following points when coding interrupt handlers.

- Coding method
  Code interrupt handlers using C or assembly language.
  When coding in C, they can be coded in the same manner as ordinary functions coded.
  When coding in assembly language, code them according to the calling rules prescribed in the compiler used.

- Stack switching
  The RI850V4 switches to the system stack specified in Basic information when passing control to an interrupt handler, and switches to the relevant stack when returning control to the processing program for which a base clock timer interrupt occurred. Coding regarding stack switching is therefore not required in interrupt handler processing.

- Service call issue
  The RI850V4 handles the interrupt handler as a "non-task".
  Service calls that can be issued in interrupt handlers are limited to the service calls that can be issued from non-tasks.

  Note 1 If a service call (isig_sem, iset_flg, etc.) accompanying dispatch processing (task scheduling processing) is issued in order to quickly complete the processing in the interrupt handler during the interval until the processing in the interrupt handler ends, the RI850V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until a return instruction is issued by the interrupt handler, upon which the actual dispatch processing is performed in batch.

  Note 2 For details on the valid issue range of each service call, refer to Table 16-1 to Table 16-12.

- Acceptance of EI level maskable interrupts
  When passing control to an interrupt handler, the RI850V4 disables acceptance of EI level maskable interrupts by manipulating the PM*n* bits (set to enabled) in the priority mask register (PMR) and the ID bit (set to disabled) in the program status word (PSW).

10.3.3 Define interrupt handler

The RI850V4 supports the static registration of interrupt handlers only. They cannot be registered dynamically by issuing a service call from the processing program.

Static interrupt handler registration means defining of interrupt handlers using static API "DEF_INH" in the system configuration file.

For details about the static API "DEF_INH", refer to "17.5.10 Interrupt handler information".

10.4 Base Clock Timer Interrupts

The RI850V4 realizes the TIME MANAGEMENT FUNCTIONS by using base clock timer interrupts that occur at constant intervals.

If a base clock timer interrupt occurs, The RI850V4's time management interrupt handler is activated and executes time-related processing (system time update, delayed wakeup/timeout of task, cyclic handler activation, etc.).

Note If acknowledgment of the relevant base clock timer interrupt is disabled by issuing loc_cpu, iloc_cpu or dis_int, the TIME MANAGEMENT FUNCTIONS may no longer operate normally.

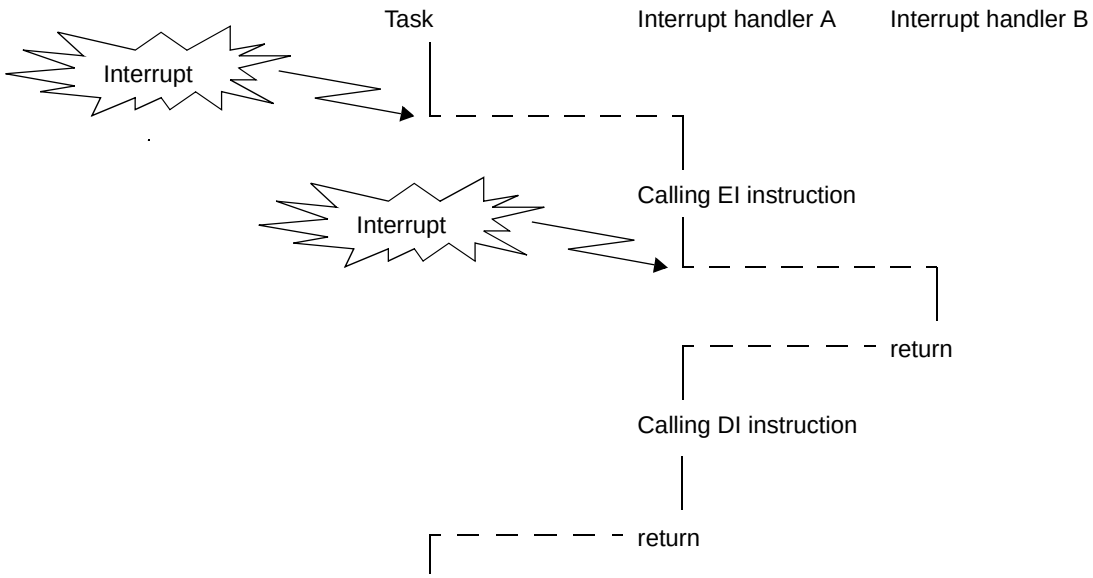
10.5 Multiple Interrupts

In the RI850V4, occurrence of an interrupt in an interrupt handler is called "multiple interrupts".

Execution of interrupt handler is started in the interrupt disabled state (the ID flag of the program status word PSW is set to 1). To generate multiple interrupts, processing to cancel the interrupt disabled state (such as issuing of EI instruction) must therefore be coded in the interrupt handler explicitly.

The following shows a processing flow when multiple interrupts occur.

Figure 10-2 Multiple Interrupts



CHAPTER 11 SERVICE CALL MANAGEMENT FUNCTIONS

This chapter describes the service call management functions performed by the RI850V4.

11.1 Outline

The RI850V4's service call management function provides the function for manipulating the extended service call routine status, such as registering and calling of extended service call routines.

11.2 Extended Service Call Routines

This is a routine to which user-defined functions are registered in the RI850V4, and will never be executed unless it is called explicitly, using service calls provided by the RI850V4.

The RI850V4 positions extended service call routines as extensions of the processing program that called the extended service call routine.

The RI850V4 manages interrupt handlers themselves, by using management objects (extended service call routine control blocks) corresponding to extended service call routines one-to-one.

11.2.1 Basic form extended service call routines

Code extended service call routines by using the ER_UINT type argument that has three VP_INT type arguments. Transferred data specified when a call request (cal_svc or ical_svc) is issued is set to arguments *par1*, *par2*, and *par3*. The following shows the basic form of extended service call routines in C.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

ER_UINT svcrtn (VP_INT par1, VP_INT par2, VP_INT par3)
{
    .........
    .........

    return (ER_UINT ercd);          /*Terminate extended service call routine*/
}
```

11.2.2 Internal processing of extended service call routine

The RI850V4 executes the original extended service call routine pre-processing when passing control from the processing program that issued a call request to an extended service call routine, as well as the original extended service call routine post-processing when returning control from the extended service call routine to the processing program.

Therefore, note the following points when coding extended service call routines.

- Coding method
  Code extended service call routines using C or assembly language.
  When coding in C, they can be coded in the same manner as ordinary functions coded.
  When coding in assembly language, code them according to the calling rules prescribed in the compiler used.

- Stack switching
  The RI850V4 positions extended service call routines as extensions of the processing program that called the extended service call routine. When passing control to an extended service call routine, stack switching processing is therefore not performed.

- Service call issue
  The RI850V4 positions extended service call routines as extensions of the processing program that called the extended service call routine. Service calls that can be issued in extended service call routines depend on the type (task or non-task) of the processing program that called the extended service call routine.

  Note For details on the valid issue range of each service call, refer to Table 16-1 to Table 16-12.

- Acceptance of EI level maskable interrupts
  The RI850V4 handles an extended service call routine as an extension of the processing program that called the extended service call routine.
  Therefore, when passing control to an extended service call routine, manipulation related to acceptance of EI level maskable interrupts (manipulation of the PM*n* bits in the priority mask register (PMR) and the ID bit in the program status word (PSW)) is not performed.

11.3 Define Extended Service Call Routine

The RI850V4 supports the static registration of extended service call routines only. They cannot be registered dynamically by issuing a service call from the processing program.

Static extended service call routine registration means defining of extended service call routines using static API "CRE_SVC" in the system configuration file.

For details about the static API "DEF_SVC", refer to "17.5.11 Extended service call routine information".

11.4 Invoke Extended Service Call Routine

Extended service call routines can be called by issuing the following service call from the processing program.

- cal_svc, ical_svc
  These service calls call the extended service call routine specified by parameter *fncd*.
  The following describes an example for coding this service call.

```
#include    <kernel.h>              /*Standard header file definition*/
#include    <kernel_id.h>           /*System information header file definition*/

void task (VP_INT exinf)
{
    ER_UINT ercd;                   /*Declares variable*/
    FN      fncd = 1;               /*Declares and initializes variable*/
    VP_INT  par1 = 123;             /*Declares and initializes variable*/
    VP_INT  par2 = 456;             /*Declares and initializes variable*/
    VP_INT  par3 = 789;             /*Declares and initializes variable*/

    .........
    .........

                                    /*Invoke extended service call routine*/
    ercd = cal_svc (fncd, par1, par2, par3);

    if (ercd != E_RSFN) {
        .........                   /*Normal termination processing*/
        .........
    }

    .........
    .........
}
```

Note Extended service call routines that can be called using this service call are the routines whose transferred data total is less than four.

CHAPTER 12 SYSTEM CONFIGURATION MANAGEMENT FUNCTIONS

This chapter describes the system configuration management functions performed by the RI850V4.

12.1 Outline

The RI850V4 provides as system configuration management functions related to the initialization routine called from Kernel Initialization Module.

12.2 User-Own Coding Module

To support various execution environments, the RI850V4 extracts from the system management functions the hardware-dependent processing (Initialization routine) that is required to execute processing, as a user-own coding module. This enhances portability for various execution environments and facilitates customization as well.

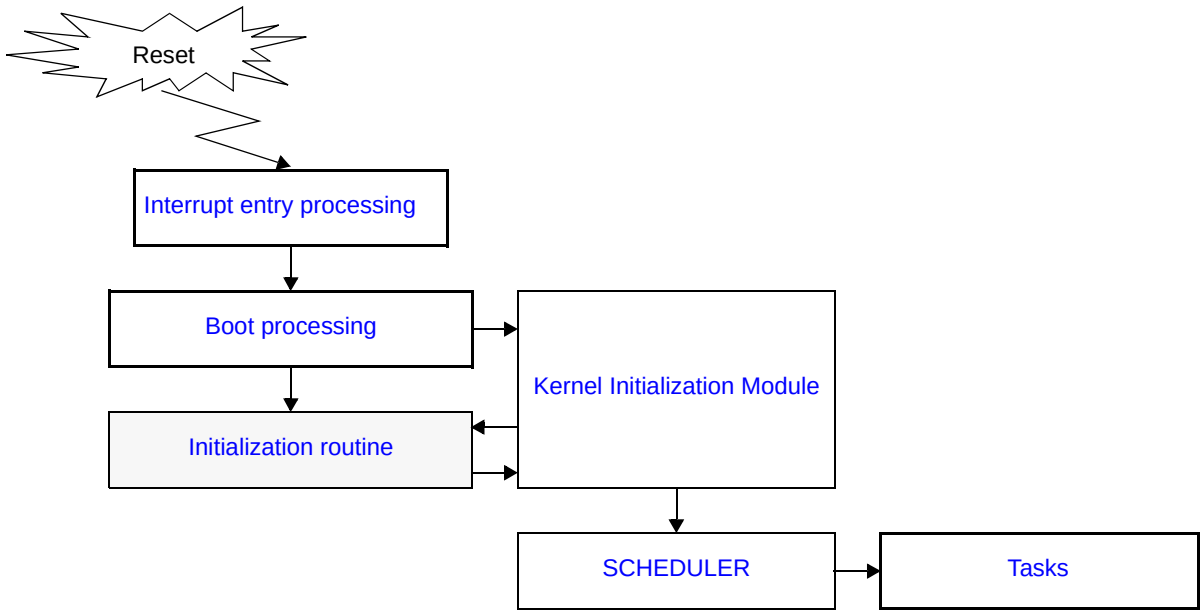
12.2.1 Initialization routine

The initialization routine is a routine dedicated to initialization processing that is extracted as a user-own coding module to initialize the hardware dependent on the user execution environment (such as the peripheral controller), and is called from the Kernel Initialization Module.

The RI850V4 manages the states in which each initialization routine may enter and initialization routines themselves, by using management objects (initialization routine control blocks) corresponding to initialization routines one-to-one.

The following shows a processing flow from when a reset interrupt occurs until the control is passed to the task.

Figure 12-1 Processing Flow (Initialization Routine)



- Basic form of initialization routines
  Code initialization routines by using the void type function that has one VP_INT type argument.

Extended information specified in Initialization routine information is set to argument *exinf*.
The following shows the basic form of initialization routine in C.

```
#include    <kernel.h>              /*Standard header file definition*/

void inirtn (VP_INT exinf)
{
    .........
    .........

    return;                         /*Terminate initialization routine*/
}
```

- Internal processing of initialization routine
  The RI850V4 executes the original initialization routine pre-processing when passing control from the Kernel Initialization Module to an initialization routine, as well as the original initialization routine post-processing when returning control from the initialization routine to the Kernel Initialization Module.
  Therefore, note the following points when coding initialization routines.

  - Coding method
    Code initialization routines using C or assembly language.
    When coding in C, they can be coded in the same manner as ordinary functions coded.
    When coding in assembly language, code them according to the calling rules prescribed in the compiler used.

  - Stack switching
    The RI850V4 switches to the system stack specified in Basic information when passing control to an initialization routine, and switches to the relevant stack when returning control to the Kernel Initialization Module. Coding regarding stack switching is therefore not required in initialization routines.

  - Service call issue
    The RI850V4 prohibits issue of service calls in Initialization routines.

  - Acceptance of EI level maskable interrupts
    When passing control to the initialization routine, the RI850V4 disables acceptance of EI level maskable interrupts by manipulating the PM*n* bits in the priority mask register (PMR) and the ID bit in the program status word (PSW).
    The PM*n* bits to be manipulated correspond to the interrupt priority range defined as the Maximum interrupt priority: maxintpri during configuration.

    Note As the RI850V4 initialization processing is not completed, acceptance is disabled for EI level maskable interrupts corresponding to the Base clock timer exception code: tim_intno defined in the Basic information and Exception code: inhno defined in the Interrupt handler information.

Note 1 When the RI850V4 initializes the hardware (OS timer) used for time management, appropriate settings should be made so that base clock timer interrupts occur according to the Base clock interval: tim_base defined in the Basic information in the system configuration file.

Note 2 Manipulate within this routine the RINT bit in the reset vector base address (RBASE) and the RINT bit in the exception handler vector address (EBASE) to specify whether operation should be done in the reduced mode, which is necessary when using an entry file in the direct vector method.

Note 3 Manipulate within this routine the MK*n* bits (or EIMK*n* bits) in the EI level interrupt mask register (IMR*m*) to enable acceptance of EI level maskable interrupts.

12.2.2 Define initialization routine

The RI850V4 supports the static registration of initialization routines only. They cannot be registered dynamically by issuing a service call from the processing program.

Static initialization routine registration means defining of initialization routines using static API "ATT_INI" in the system configuration file.

For details about the static API "ATT_INI", refer to "17.5.12 Initialization routine information".

CHAPTER 13 SCHEDULER

This chapter describes the scheduler of the RI850V4.

13.1 Outline

The scheduling functions provided by the RI850V4 consist of functions manage/decide the order in which tasks are executed by monitoring the transition states of dynamically changing tasks, so that the CPU use right is given to the optimum task.

13.1.1 Drive Method

The RI850V4 employs the Event-driven system in which the scheduler is activated when an event (trigger) occurs.

- Event-driven system

  Under the event-driven system of the RI850V4, the scheduler is activated upon occurrence of the events listed below and dispatch processing (task scheduling processing) is executed.

  - Issue of service call that may cause task state transition
  - Issue of instruction for returning from non-task (cyclic handler, interrupt handler, etc.)
  - Occurrence of clock interrupt used when achieving TIME MANAGEMENT FUNCTIONS
  - vsta_sch issue

13.1.2 Scheduling Method

As task scheduling methods, the RI850V4 employs the Priority level method, which uses the priority level defined for each task, and the FCFS method, which uses the time elapsed from the point when a task becomes subject to RI850V4 scheduling.

- Priority level method

  A task with the highest priority level is selected from among all the tasks that have entered an executable state (RUNNING state or READY state), and given the CPU use right.

- FCFS method

  The same priority level can be defined for multiple tasks in the RI850V4. Therefore, multiple tasks with the highest priority level, which is used as the criterion for task selection under the Priority level method, may exist simultaneously.
  To remedy this, dispatch processing (task scheduling processing) is executed on a first come first served (FCFS) basis, and the task for which the longest interval of time has elapsed since it entered an executable state (READY state) is selected as the task to which the CPU use right is granted.

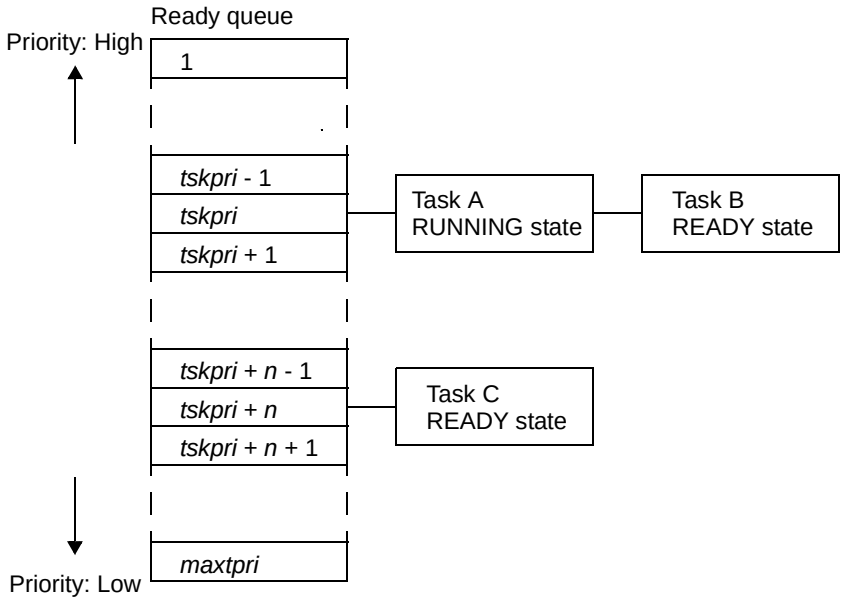
13.1.3 Ready queue

The RI850V4 uses a "ready queue" to implement task scheduling.

The ready queue is a hash table that uses priority as the key, and tasks that have entered an executable state (READY state or RUNNING state) are queued in FIFO order. Therefore, the scheduler realizes the RI850V4's scheduling method (priority level or FCFS) by executing task detection processing from the highest priority level of the ready queue upon activation, and upon detection of queued tasks, giving the CPU use right to the first task of the proper priority level.

The following shows the case where multiple tasks are queued to a ready queue.

Figure 13-1 Implementation of Scheduling Method (Priority Level Method or FCFS Method)



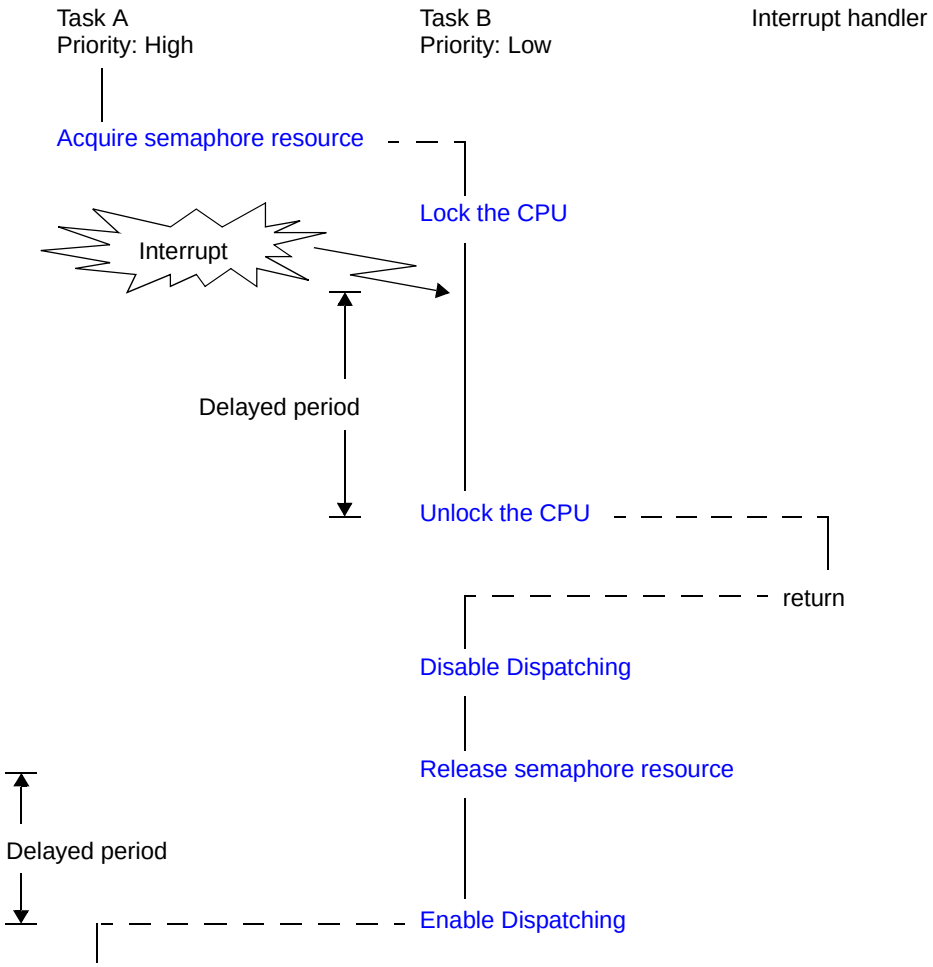
- Create ready queue
  In the RI850V4, the method of creating a ready queue is limited to "static creation".
  Ready queues therefore cannot be created dynamically using a method such as issuing a service call from a processing program.
  Static ready queue creation means defining of maximum priority using static API "MAX_PRI" in the system configuration file.
  For details about the basic information "MAX_PRI", refer to "17.4.2 Basic information".

13.1.4 Scheduling Lock Function

The RI850V4 provides the scheduling lock function for manipulating the scheduler status explicitly from the processing program and disabling/enabling dispatch processing.

The following shows a processing flow when using the scheduling lock function.

Figure 13-2 Scheduling Lock Function



The scheduling lock function can be implemented by issuing the following service call from the processing program.

loc_cpu, iloc_cpu, unl_cpu, iunl_cpu, dis_dsp, ena_dsp

13.2 User-Own Coding Module

To support various execution environments, the hardware-dependent processing (idle routine) that is required for the RI850V4 to execute processing is extracted from the scheduling facility as a user-own coding module.

This enhances portability to various execution environments and facilitates customization as well.

13.2.1 Idle Routine

The idle routine is a routine dedicated to idle processing that is extracted as a user-own coding module to utilize the standby function provided by the CPU (to achieve the low-power consumption system), and is called from the scheduler when there no longer remains a task subject to scheduling by the RI850V4 (task in the RUNNING or READY state) in the system.

The RI850V4 manages the states in which each idle routine may enter and idle routines themselves, by using management objects (idle routine control blocks) corresponding to idle routines one-to-one.

- Basic form of idle routine

Code idle routines by using the void type function that has no arguments.

The following shows the basic form of idle routine in C.

```
#include    <kernel.h>              /*Standard header file definition*/

void idlrtn (void)
{
    /* ......... */

    return;                         /*Terminate idle routine*/
}
```

- Internal processing of idle routine

The RI850V4 executes "original pre-processing" when passing control to the idle routine, as well as "original post-processing" when regaining control from the idle routine.

Therefore, note the following points when coding idle routines.

- Coding method
  Code idle routines using C or assembly language.
  When coding in C, they can be coded in the same manner as ordinary functions coded.
  When coding in assembly language, code them according to the calling rules prescribed in the compiler used.
- Stack switching
  The RI850V4 switches to the system stack specified in Basic information when passing control to an idle routine.
  Coding regarding stack switching is therefore not required in idle routines.
- Service call issue
  The RI850V4 prohibits issue of service calls in idle routines.
- Acceptance of EI level maskable interrupts
  When passing control to the idle routine, the RI850V4 enables acceptance of EI level maskable interrupts by manipulating the PM*n* bits in the priority mask register (PMR) and the ID bit in the program status word (PSW).
  The PM*n* bits to be manipulated correspond to the interrupt priority range defined as the Maximum interrupt priority: maxintpri during configuration.

  Note In most cases, control returns from the idle routine (moves to another processing program) if the wait time has passed or an EI level maskable interrupt occurs, do not issue the DI instruction in the idle routine.

13.2.2 Define Idle Routine

The RI850V4 supports the static registration of idle routines only. They cannot be registered dynamically by issuing a service call from the processing program.

Static idle routine registration means defining of idle routines using static API "VATT_IDL" in the system configuration file.

For details about the static API "VATT_IDL", refer to "17.5.13 Idle routine information".

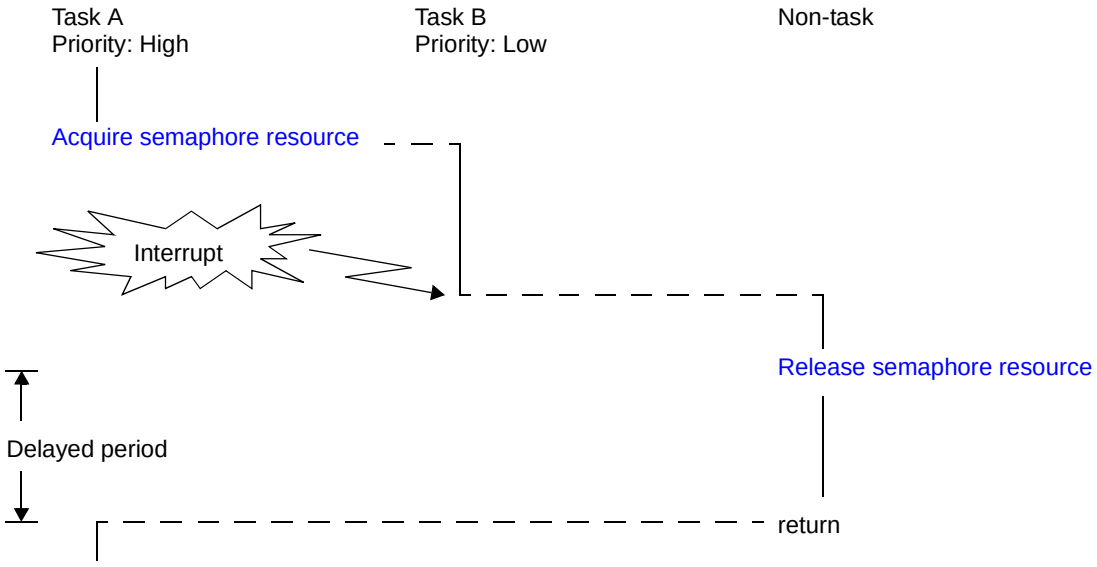
Note If Idle routine information is not defined, the default idle routine (function name: _kernel_default_idlrtn) is registered during configuration.
The default idle routine issues the HALT instruction.

13.3 Scheduling in Non-Tasks

If a service call (isig_sem, iset_flg, etc.) accompanying dispatch processing (task scheduling processing) is issued in order to quickly complete the processing in the non-task (cyclic handler, interrupt handler, etc.) during the interval until the processing in the non-task ends, the RI850V4 executes only processing such as queue manipulation and the actual dispatch processing is delayed until a return instruction is issued, upon which the actual dispatch processing is performed in batch.

The following shows a processing flow when a service call accompanying dispatch processing is issued in a non-task.

Figure 13-3 Scheduling in Non-Tasks



CHAPTER 14 SYSTEM INITIALIZATION ROUTINE

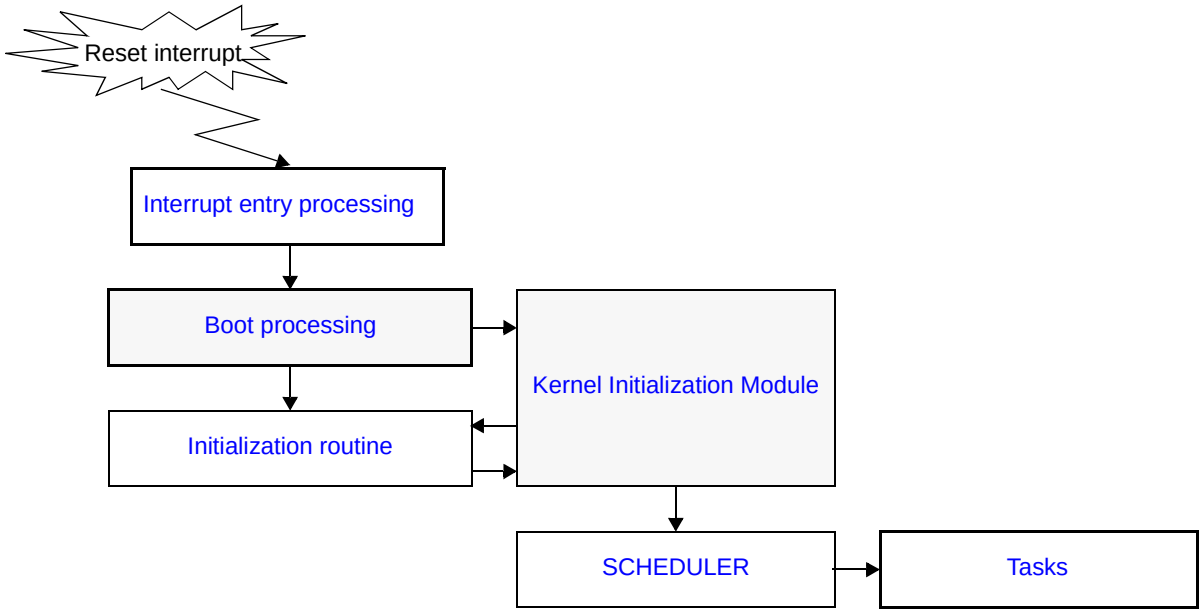
This chapter describes the system initialization routine performed by the RI850V4.

14.1 Outline

The system initialization routine of the RI850V4 provides system initialization processing, which is required from the reset interrupt output until control is passed to the task.

The following shows a processing flow from when a reset interrupt occurs until the control is passed to the task.

Figure 14-1 Processing Flow (System Initialization)



14.2 User-Own Coding Module

To support various execution environments, the RI850V4 extracts from the system initialization processing the hardware-dependent processing (Boot processing) that is required to execute processing, as a user-own coding module. This enhances portability for various execution environments and facilitates customization as well.

14.2.1 Boot processing

This is a routine dedicated to initialization processing that is extracted as a user-own coding module to initialize the minimum required hardware for the RI850V4 to perform processing, and is called from Initialization routine.

- Basic form of boot processing
  Code boot processing by using the void type function that has no arguments.
  The following shows the basic form of boot processing in assembly.

```
    .public __boot

    .text   .cseg   text
    .align  0x2
__boot :

    .........
    .........

    mov     #__kernel_start, r11       /*Jump to Kernel Initialization Module*/
    jarl    [r11], lp
```

- Internal processing of boot processing
  Boot processing is a routine dedicated to initialization processing that is called from Initialization routine, without RI850V4 intervention.
  Therefore, note the following points when coding boot processing.
  - Coding method
    Code boot processing using C or assembly language.
    When coding in C, they can be coded in the same manner as ordinary functions coded.
    When coding in assembly language, code them according to the calling rules prescribed in the compiler used.
  - Stack switching
    Setting of stack pointer SP is not executed at the point when control is passed to boot processing.
    To use a boot processing dedicated stack, setting of stack pointer SP must therefore be coded at the beginning of the boot processing.
  - Service call issue
    Execution of the Kernel Initialization Module is not performed when boot processing is started. issue of service calls is therefore prohibited during boot processing.

The following lists processing that should be executed in boot processing.

1) Initializing the interrupt configuration register (INTCFG)

2) Setting of global pointer GP

3) Setting of element pointer EP

4) Setting stack pointer SP

5) Setting of text pointer TP (only when using the CCV850 compiler)

6) Initialization of ".kernel_data_init" section

7) Setting of the interrupt priority (manipulation of EIC.EIPn)

8) Selection of the interrupt vector method (manipulation of EIC.EITB)

9) Initialization of memory areas for uninitialized data (such as the bss section)

10) Initialization of Internal unit of CPU (such as the OS timer) and peripheral controllers (For initialization of the OS timer, see "9.2.1 Base clock timer interrupt".

11) Setting of the priority for the floating-point operation exception (only for the device incorporating the FPU)

12) Setting of the start address of the system information table in *r6*

13) Returning of control to the Kernel Initialization Module _kernel_start

Note 1 When initializing the interrupt configuration register (INTCFG), clear the ISPC bit to 0 to automatically update the ISPR value.

Note 2 If global pointer GP is modified outside the boot processing, correct operation is not guaranteed afterwards.

Note 3 When modifying element pointer EP in a processing program (such as a task or a cyclic handler), compiler option –Xep = callee must be specified (Property panel -> [Common Options] tabbed page -> [Register Mode] category -> [ep-register treatment] must be set to "Treat as callee-save").

Note 4 Setting of stack pointer sp is required only when the dedicated boot processing stack is used.

Note 5 The following shows sample codes for initializing ".kernel_data_init" section.

- CC-RH compiler

  Use the RAM section area initialization function _INITSCT_RH in the CC-RH compiler to simplify the initialization code.

  The following shows an example of assembly-language code for initializing ".kernel_data_init" section.

```
.section    ".INIT_BSEC.const", const
.align      0x4
.dw         #__s.kernel_data_init, #__e.kernel_data_init
mov         r0, r6
mov         r0, r7
mov         #__s.INIT_BSEC.const, r8
mov         #__e.INIT_BSEC.const, r9
jarl        __INITSCT_RH, lp
```

- CCV850 compiler

  First, define ".kernel_data_init" section (where the kernel initialization flag is allocated) with the CLEAR section attribute added in the link directive file so that the section is initialized in the boot processing.

  The following shows a sample code in the link directive file.

```
MEMORY {
ROM_MEMORY: ORIGIN = 0x00007000,LENGTH = 0x003f9000
RAM_MEMORY: ORIGIN = 0xfedf0000,LENGTH = 0x00010000
    ............
}
SECTIONS {
    ............
.kernel_data_init  CLEAR:>RAM_MEMORY
    ............
}
```

Withthis definition, the ".kernel_data_init" section information is included in the runtime clear tables (__ghsbinfo_clear and __ghseinfo_clear) generated by the CCV850 compiler. The following shows a sample code for initializing the section area included in the runtime clear tables.

```
/* initialize memory */
meminit(void){
 {
 void **b = (void **)__ghsbinfo_clear;
 void **e = (void **)__ghseinfo_clear;

 while (b < e){
  void *s, *n, *v;

  s = (sint8 *)(*b++);
  v = *b++;
  n = *b++;
  memset(s, (sint32)v, (uint32)n);
  }
 }
.............
```

Note 6 Manipulate within the Initialization routine the RINT bit in the reset vector base address (RBASE) and the RINT bit in the exception handler vector address (EBASE) to specify whether operation should be done in the reduced mode, which is necessary when using an entry file in the direct vector method.

Note 7 Manipulate within the Initialization routine the MK*n* bits (or EIMK*n* bits) in the EI level interrupt mask register (IMR*m*) to enable acceptance of EI level maskable interrupts.

Note 8 Set the priority of the floating-point operation exception to a higher value than the maximum priority of interrupts managed by the kernel.

14.2.2 System dependent information

System-dependent information is a header file (file name: userown.h) including various information that is required for the RI850V4, which is extracted as a user-own coding module.

- Basic form of system-dependent information
  When coding system-dependent information, use a specified file name (userown.h) and specified macro names (KERNEL_USR_TMCNTREG and KERNEL_USR_BASETIME).

  The following shows the basic form of system-dependent information in the C language.

```
#include    <kernel_id.h>              /*System information header file definition*/

#define     KERNEL_USR_TMCNTREG 0xffec0004  /*I/O address*/

#define     KERNEL_USR_BASETIME 250         /*time for one count (4MHz -> 250ns)*/
```

The following is a list of information that should be defined as system-dependent information.

- Definition of the system information header file
  Include the system information header file output as a result of execution of the configurator for the system configuration file.

  Note This information is necessary only when "Taking in long-statistics by software trace mode" is selected in the Property panel -> [Task Analyzer] tabbed page -> [Trace] category -> [Selection of trace mode].

- Basic clock timer information
  Define as macros the I/O address (register base address + 0x4) for the counter register (OSTMn) and the time (in nanoseconds) for one count in the OS timer calculated according to the frequency of the OS timer operation.

Note This information is necessary only when "Taking in trace chart by software trace mode" or "Taking in long-statistics by software trace mode" is selected in the Property panel -> [Task Analyzer] tabbed page -> [Trace] category -> [Selection of trace mode].

14.3 Kernel Initialization Module

The kernel initialization module is a dedicated initialization processing routine provided for initializing the minimum required software for the RI850V4 to perform processing, and is called from Boot processing.

The following processing is executed in the kernel initialization module.

- Initializing management objects
  Initializes the objects defined in the system configuration file (such as tasks and semaphores).

- Initializing the system time
  Initializes the system time (to 0), which is updated in units of the Base clock interval: tim_base when an EI level maskable interrupt defined in the Base clock timer exception code: tim_intno occurs.

- Activating tasks
  Moves the tasks whose Attribute: tskatr (such as coding language and initial activation state) is defined as TA_ACT from the DORMANT state to the READY state.

- Starting cyclic handlers
  Moves the cyclic handlers whose Attribute: cycatr (such as coding language and initial activation state) is defined as TA_STA from the non-operational state (STP state) to the operational state (STA state).

- Calling initialization routines
  Calls the initialization routines defined in the Initialization routine information in the order of definitions in the system configuration file.

- Returning control to the scheduler
  Selects the most suitable one of the tasks placed in the READY state and moves the task from the READY state to the RUNNING state.

Note The kernel initialization module is included in the system initialization processing provided by the RI850V4. The user is not required to code the kernel initialization module.

CHAPTER 15 DATA TYPES AND MACROS

This chapter describes the data types, data structures and macros, which are used when issuing service calls provided by the RI850V4.

The definition of the macro and data structures is performed by each header file stored in <ri_root>\include\os.

Note <ri_root> indicates the installation folder of RI850V4.
The default folder is “C:\Program Files\Renesas Electronics\CS+\CC\RI850V4RH.

15.1 Data Types

The Following lists the data types of parameters specified when issuing a service call.

Macro definition of the data type is performed by the header file <ri_root>\include\os\types.h, which is called from the standard header file <ri_root>\include\kernel.h and the ITRON general definition header file <ri_root>\include\os\itron.h.

Table 15-1 Data Types

Macro	Data Type	Description
B	signed char	Signed 8-bit integer
H	signed short	Signed 16-bit integer
W	signed long	Signed 32-bit integer
UB	unsigned char	Unsigned 8-bit integer
UH	unsigned short	Unsigned 16-bit integer
UW	unsigned long	Unsigned 32-bit integer
VB	signed char	8-bit value with unknown data type
VH	signed short	16-bit value with unknown data type
VW	signed long	32-bit value with unknown data type
VP	void *	Pointer to unknown data type
FP	void (*)	Processing unit start address (pointer to a function)
INT	signed int	Signed 32-bit integer
UINT	unsigned int	Unsigned 32-bit integer
BOOL	signed long	Boolean value (TRUE or FALSE)
FN	signed short	Function code
ER	signed long	Error code
ID	signed short	Object ID number
ATR	unsigned short	Object attribute
STAT	unsigned short	Object state
MODE	unsigned short	Service call operational mode
PRI	signed short	Priority
SIZE	unsigned long	Memory area size (in bytes)
TMO	signed long	Timeout (unit:millisecond)
RELTIM	unsigned long	Relative time (unit:millisecond)
VP_INT	signed int	Pointer to unknown data type, or signed 32-bit integer

Macro	Data Type	Description
ER_BOOL	signed long	Error code, or boolean value (TRUE or FALSE)
ER_ID	signed long	Error code, or object ID number
ER_UINT	signed int	Error code, or signed 32-bit integer
FLGPTN	unsigned int	Bit pattern
INTNO	unsigned short	Exception code

15.2 Packet Formats

This section explains the data structures (task state packet, semaphore state packet, or the like) used when issuing a service call provided by the RI850V4.

Be sure not to refer from programs to the areas reserved for future use in each data structure.

15.2.1 Task state packet

The following shows task state packet T_RTSK used when issuing ref_tsk or iref_tsk.

Definition of task state packet T_RTSK is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rtsk {
    STAT    tskstat;        /*Current state*/
    PRI     tskpri;         /*Current priority*/
    PRI     tskbpri;        /*Reserved for future use*/
    STAT    tskwait;        /*Reason for waiting*/
    ID      wobjid;         /*Object ID number for which the task waiting*/
    TMO     lefttmo;        /*Remaining time until timeout*/
    UINT    actcnt;         /*Activation request count*/
    UINT    wupcnt;         /*Wakeup request count*/
    UINT    suscnt;         /*Suspension count*/
    ATR     tskatr;         /*Attribute*/
    PRI     itskpri;        /*Initial priority*/
    ID      memid;          /*Reserved for future use*/
} T_RTSK;
```

The following shows details on task state packet T_RTSK.

- tskstat
  Stores the current state.

TTS_RUN:	RUNNING state
TTS_RDY:	READY state
TTS_WAI:	WAITING state
TTS_SUS:	SUSPENDED state
TTS_WAS:	WAITING-SUSPENDED state
TTS_DMT:	DORMANT state

- tskpri
  Stores the current priority.

- tskbpri
  System-reserved area.

- tskwait
  Stores the reason for waiting.

TTW_SLP:	Sleeping state
TTW_DLY:	Delayed state
TTW_SEM:	WAITING state for a semaphore resource
TTW_FLG:	WAITING state for an eventflag
TTW_SDTQ:	Sending WAITING state for a data queue
TTW_RDTQ:	Receiving WAITING state for a data queue
TTW_MBX:	Receiving WAITING state for a mailbox
TTW_MTX:	WAITING state for a mutex
TTW_MPF:	WAITING state for a fixed-sized memory block
TTW_MPL:	WAITING state for a variable-sized memory block

- wobjid
  Stores the object ID number for which the task waiting.
  When the task is not in the WAITING state, 0 is stored.

- lefttmo
  Stores the remaining time until timeout (unit:millisecond).

- actcnt
  Stores the activation request count.

- wupcnt
  Stores the wakeup request count.

- suscnt
  Stores the suspension count.

- tskatr
  Stores the attribute (coding language, initial activation state, etc.).

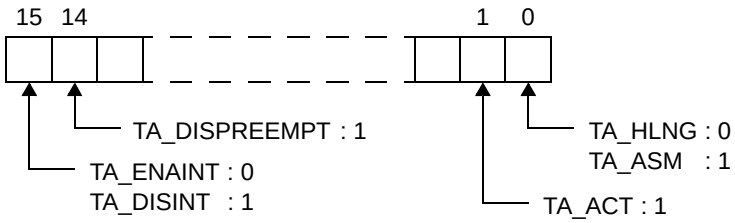
  Coding language (bit 0)
  TA_HLNG: Start a task through a C language interface.
  TA_ASM: Start a task through an assembly language interface.

  Initial activation state (bit 1)
  TA_ACT: Task is activated after the creation.

  Initial preemption state (bit 14)
  TA_DISPREEMPT: Preemption is disabled at task activation.

  Initial interrupt state (bit 15)
  TA_ENAINT: Acceptance of EI level maskable interrupts (from the Maximum interrupt priority: maxintpri to the minimum interrupt priority) is enabled.
  TA_DISINT: Acceptance of EI level maskable interrupts (from the Maximum interrupt priority: maxintpri to the minimum interrupt priority) is disabled.

  [Structure of tskatr]



- itskpri
  Stores the initial priority.

- memid
  System-reserved area.

15.2.2 Task state packet (simplified version)

The following shows task state packet (simplified version) T_RTST used when issuing ref_tst or iref_tst.

Definition of task state packet (simplified version) T_RTST is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rtst {
    STAT    tskstat;        /*Current state*/
    STAT    tskwait;        /*Reason for waiting*/
} T_RTST;
```

The following shows details on task state packet (simplified version) T_RTST.

- tskstat
  Stores the current state.

TTS_RUN:	RUNNING state
TTS_RDY:	READY state
TTS_WAI:	WAITING state
TTS_SUS:	SUSPENDED state
TTS_WAS:	WAITING-SUSPENDED state
TTS_DMT:	DORMANT state

- tskwait
  Stores the reason for waiting.

TTW_SLP:	Sleeping state
TTW_DLY:	Delayed state
TTW_SEM:	WAITING state for a semaphore resource
TTW_FLG:	WAITING state for an eventflag
TTW_SDTQ:	Sending WAITING state for a data queue
TTW_RDTQ:	Receiving WAITING state for a data queue
TTW_MBX:	Receiving WAITING state for a mailbox
TTW_MTX:	WAITING state for a mutex
TTW_MPF:	WAITING state for a fixed-sized memory block
TTW_MPL:	WAITING state for a variable-sized memory block

15.2.3 Semaphore state packet

The following shows semaphore state packet T_RSEM used when issuing ref_sem or iref_sem.

Definition of semaphore state packet T_RSEM is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rsem {
    ID      wtskid;         /*Existence of waiting task*/
    UINT    semcnt;         /*Current resource count*/
    ATR     sematr;         /*Attribute*/
    UINT    maxsem;         /*Maximum resource count*/
} T_RSEM;
```

The following shows details on semaphore state packet T_RSEM.

- wtskid
  Stores whether a task is queued to the semaphore wait queue.

TSK_NONE:	No applicable task
Value:	ID number of the task at the head of the wait queue

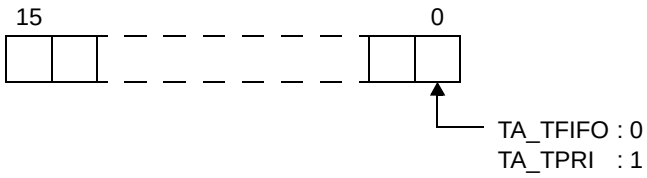
- semcnt
  Stores the current resource count.

- sematr
  Stores the attribute (queuing method).

  Task queuing method (bit 0)

TA_TFIFO:	Task wait queue is in FIFO order.
TA_TPRI:	Task wait queue is in task priority order.

  [Structure of sematr]



- maxsem
  Stores the maximum resource count.

15.2.4 Eventflag state packet

The following shows eventflag state packet T_RFLG used when issuing ref_flg or iref_flg.

Definition of eventflag state packet T_RFLG is performed by the header file <ri_root>\include\os\packet.h, which is called the from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rflg {
    ID      wtskid;         /*Existence of waiting task*/
    FLGPTN  flgptn;         /*Current bit pattern*/
    ATR     flgatr;         /*Attribute*/
} T_RFLG;
```

The following shows details on eventflag state packet T_RFLG.

- wtskid
  Stores whether a task is queued to the event flag wait queue.

TSK_NONE:	No applicable task
Value:	ID number of the task at the head of the wait queue

- flgptn
  Stores the Current bit pattern.

- flgatr
  Stores the attribute (queuing method, queuing count, etc.).

  Task queuing method (bit 0)

TA_TFIFO:	Task wait queue is in FIFO order.
TA_TPRI:	Task wait queue is in task priority order.

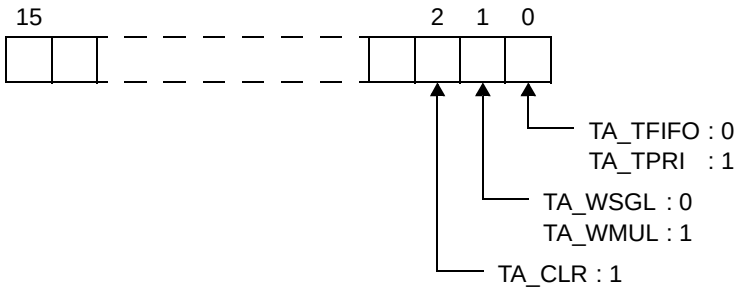
  Queuing count (bit 1)

TA_WSGL:	Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL:	Multiple tasks are allowed to be in the WAITING state for the eventflag.

  Bit pattern clear (bit 2)

TA_CLR:	Bit pattern is cleared when a task is released from the WAITING state for eventflag.

  [Structure of flgatr]



15.2.5 Data queue state packet

The following shows data queue state packet T_RDTQ used when issuing ref_dtq or iref_dtq.

Definition of data queue state packet T_RDTQ is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rdtq {
    ID      stskid;         /*Existence of tasks waiting for data transmission*/
    ID      rtskid;         /*Existence of tasks waiting for data reception*/
    UINT    sdtqcnt;        /*number of data elements in the data queue*/
    ATR     dtqatr;         /*Attribute*/
    UINT    dtqcnt;         /*Data count*/
    ID      memid;          /*Reserved for future use*/
} T_RDTQ;
```

The following shows details on data queue state packet T_RDTQ.

- stskid
  Stores whether a task is queued to the transmission wait queue of the data queue.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

- rtskid
  Stores whether a task is queued to the reception wait queue of the data queue.

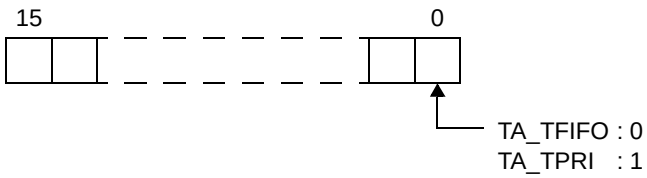
  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

- sdtqcnt
  Stores the number of data elements in data queue.

- dtqatr
  Stores the attribute (queuing method).

  Task queuing method (bit 0)
  TA_TFIFO: Task wait queue is in FIFO order.
  TA_TPRI: Task wait queue is in task priority order.

  [Structure of dtqatr]



- dtqcnt
  Stores the data count.

- memid
  System-reserved area.

15.2.6 Message packet

The following shows message packet T_MSG/T_MSG_PRI used when issuing snd_mbx, isnd_mbx, rcv_mbx, prcv_mbx, iprcv_mbx or trcv_mbx.

Definition of message packet T_MSG/T_MSG_PRI is performed by the header file <ri_root>\include\os\types.h, which is called from the standard header file <ri_root>\include\kernel.h and the ITRON general definition header file <ri_root>\include\itron.h.

[Message packet for TA_MFIFO attribute]

```
typedef struct  t_msg {
    struct  t_msg   *msgnext;   /*Reserved for future use*/
} T_MSG;
```

[Message packet for TA_MPRI attribute]

```
typedef struct  t_msg_pri {
    struct  t_msg   msgque;     /*Reserved for future use*/
    PRI     msgpri;             /*Message priority*/
} T_MSG_PRI;
```

The following shows details on message packet T_RTSK/T_MSG_PRI.

- msgnext, msgque
  System-reserved area.

- msgpri
  Stores the message priority.

  Note 1 In the RI850V4, a message having a smaller priority number is given a higher priority.

  Note 2 Values that can be specified as the message priority level are limited to the range defined in Mailbox information (Maximum message priority: maxmpri) when the system configuration file is created.

15.2.7 Mailbox state packet

The following shows mailbox state packet T_RMBX used when issuing ref_mbx or iref_mbx.

Definition of mailbox state packet T_RMBX is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rmbx {
    ID      wtskid;         /*Existence of waiting task*/
    T_MSG   *pk_msg;        /*Existence of waiting message*/
    ATR     mbxatr;         /*Attribute*/
} T_RMBX;
```

The following shows details on mailbox state packet T_RMBX.

- wtskid
  Stores whether a task is queued to the mailbox wait queue.

TSK_NONE:	No applicable task
Value:	ID number of the task at the head of the wait queue

- pk_msg
  Stores whether a message is queued to the mailbox wait queue.

NULL:	No applicable message
Value:	Start address of the message packet at the head of the wait queue

- mbxatr
  Stores the attribute (queuing method).

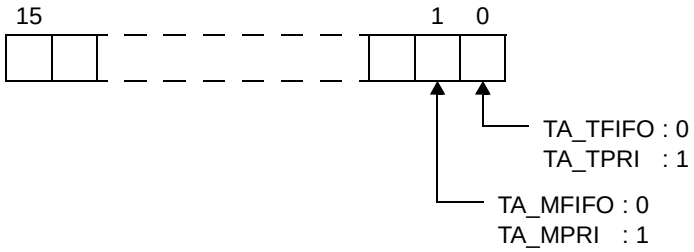
  Task queuing method (bit 0)

TA_TFIFO:	Task wait queue is in FIFO order.
TA_TPRI:	Task wait queue is in task priority order.

  Message queuing method (bit 1)

TA_MFIFO:	Message wait queue is in FIFO order.
TA_MPRI:	Message wait queue is in message priority order.

  [Structure of mbxatr]



15.2.8 Mutex state packet

The following shows mutex state packet T_RMTX used when issuing ref_mtx or iref_mtx.

Definition of mutex state packet T_RMTX is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rmtx {
    ID      htskid;         /*Existence of locked mutex*/
    ID      wtskid;         /*Existence of waiting task*/
    ATR     mtxatr;         /*Attribute*/
    PRI     ceilpri;        /*Reserved for future use*/
} T_RMTX;
```

The following shows details on mutex state packet T_RMTX.

- htskid
  Stores whether a task that is locking a mutex exists.

  TSK_NONE: No applicable task
  Value: ID number of the task locking the mutex

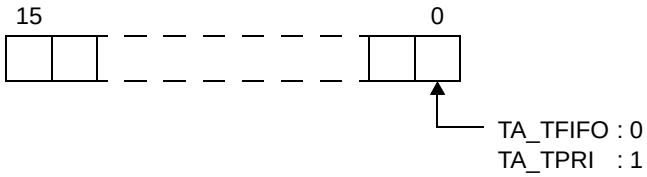
- wtskid
  Stores whether a task is queued to the mutex wait queue.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

- mtxatr
  Stores the attribute (queuing method).

  Task queuing method (bit 0 to 1)
  TA_TFIFO: Task wait queue is in FIFO order.
  TA_TPRI: Task wait queue is in task priority order.

  [Structure of mtxatr]



- ceilpri
  System-reserved area.

15.2.9 Fixed-sized memory pool state packet

The following shows fixed-sized memory pool state packet T_RMPF used when issuing ref_mpf or iref_mpf.

Definition of fixed-sized memory pool state packet T_RMPF is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rmpf {
    ID      wtskid;         /*Existence of waiting task*/
    UINT    fblkcnt;        /*Number of free memory blocks*/
    ATR     mpfatr;         /*Attribute*/
    ID      memid;          /*Reserved for future use*/
} T_RMPF;
```

The following shows details on fixed-sized memory pool state packet T_RMPF.

- wtskid
  Stores whether a task is queued to the fixed-size memory pool.

TSK_NONE:	No applicable task
Value:	ID number of the task at the head of the wait queue

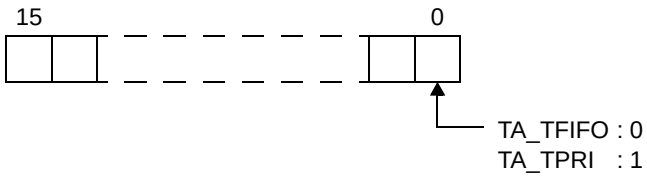
- fblkcnt
  Stores the number of free memory blocks.

- mpfatr
  Stores the attribute (queuing method).

  Task queuing method (bit 0)

TA_TFIFO:	Task wait queue is in FIFO order.
TA_TPRI:	Task wait queue is in task priority order.

  [Structure of mpfatr]



- memid
  System-reserved area.

15.2.10 Variable-sized memory pool state packet

The following shows variable-sized memory pool state packet T_RMPL used when issuing ref_mpl or iref_mpl.

Definition of variable-sized memory pool state packet T_RMPL is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rmpl {
    ID      wtskid;         /*Existence of waiting task*/
    SIZE    fmplsz;         /*Total size of free memory blocks*/
    UINT    fblksz;         /*Maximum memory block size available*/
    ATR     mplatr;         /*Attribute*/
    ID      memid;          /*Reserved for future use*/
} T_RMPL;
```

The following shows details on variable-sized memory pool state packet T_RMPL.

- wtskid
  Stores whether a task is queued to the variable-size memory pool wait queue.

  TSK_NONE: No applicable task
  Value: ID number of the task at the head of the wait queue

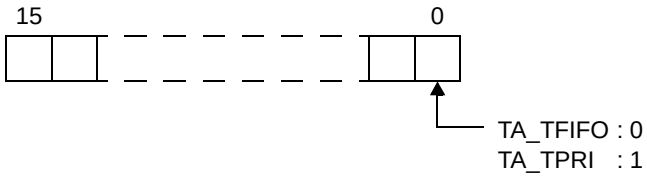
- fmplsz
  Stores the total size of free memory blocks (in bytes).

- fblksz
  Stores the maximum memory block size available (in bytes).

- mplatr
  Stores the attribute (queuing method).

  Task queuing method (bit 0)
  TA_TFIFO: Task wait queue is in FIFO order.
  TA_TPRI: Task wait queue is in task priority order.

  [Structure of mplatr]



- memid
  System-reserved area.

15.2.11 System time packet

The following shows system time packet SYSTIM used when issuing set_tim, iset_tim, get_tim or iget_tim.

Definition of system time packet SYSTIM is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_systim {
    UW      ltime;          /*System time (lower 32 bits)*/
    UH      utime;          /*System time (higher 16 bits)*/
} SYSTIM;
```

The following shows details on system time packet SYSTIM.

- ltime
  Stores the system time (lower 32 bits).

- utime
  Stores the system time (higher 16 bits).

15.2.12 Cyclic handler state packet

The following shows cyclic handler state packet T_RCYC used when issuing ref_cyc or iref_cyc.

Definition of cyclic handler state packet T_RCYC is performed by the header file <ri_root>\include\os\packet.h, which is called from the standard header file <ri_root>\include\kernel.h.

```
typedef struct  t_rcyc {
    STAT    cycstat;        /*Current state*/
    RELTIM  lefttim;        /*Time left before the next activation*/
    ATR     cycatr;         /*Attribute*/
    RELTIM  cyctim;         /*Activation cycle*/
    RELTIM  cycphs;         /*Activation phase*/
} T_RCYC;
```

The following shows details on cyclic handler state packet T_RCYC.

- cycstat
  Store the current state.

  TCYC_STP: Non-operational state
  TCYC_STA: Operational state

- lefttim
  Stores the time left before the next activation (unit:millisecond).

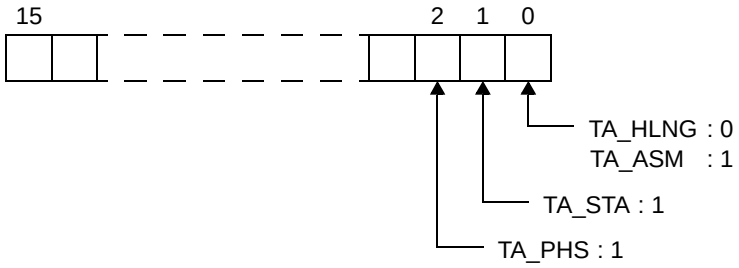
- cycatr
  Stores the attribute (coding language, initial activation state, etc.).

  Coding language (bit 0)
  TA_HLNG: Start a cyclic handler through a C language interface.
  TA_ASM: Start a cyclic handler through an assembly language interface.

  Initial activation state (bit 1)
  TA_STA: Cyclic handlers is in an operational state after the creation.

  Existence of saved activation phases (bit 2)
  TA_PHS: Cyclic handler is activated preserving the activation phase.

  [Structure of cycatr]



- cyctim
  Stores the activation cycle (unit:millisecond).

- cycphs
  Stores the activation phase (unit:millisecond).
  In the RI850V4, the initial activation phase means the relative interval from when generation of s cyclic handler is completed until the first activation request is issued.

15.3 Data Macros

This section explains the data macros (for current state, processing program attributes, or the like) used when issuing a service call provided by the RI850V4.

15.3.1 Current state

The following lists the management object current states acquired by issuing service calls (ref_tsk, ref_sem, or the like).
Macro definition of the current state is performed by the header file <ri_root>\include\os\option.h, which is called from standard the header file <ri_root>\include\kernel.h and the ITRON general definition header file <ri_root>\include\itron.h.

Table 15-2 Current State

Macro	Value	Description
TTS_RUN	0x01	RUNNING state
TTS_RDY	0x02	READY state
TTS_WAI	0x04	WAITING state
TTS_SUS	0x08	SUSPENDED state
TTS_WAS	0x0c	WAITING-SUSPENDED state
TTS_DMT	0x10	DORMANT state
TCYC_STP	0x00	Non-operational state
TCYC_STA	0x01	Operational state
TTW_SLP	0x0001	Sleeping state
TTW_DLY	0x0002	Delayed state
TTW_SEM	0x0004	WAITING state for a semaphore resource
TTW_FLG	0x0008	WAITING state for an eventflag
TTW_SDTQ	0x0010	Sending WAITING state for a data queue
TTW_RDTQ	0x0020	Receiving WAITING state for a data queue
TTW_MBX	0x0040	Receiving WAITING state for a mailbox
TTW_MTX	0x0080	WAITING state for a mutex
TTW_MPF	0x2000	WAITING state for a fixed-sized memory pool
TTW_MPL	0x4000	WAITING state for a variable-sized memory pool
TSK_NONE	0	No applicable task

15.3.2 Processing program attributes

The following lists the processing program attributes acquired by issuing service calls (ref_tsk, ref_cyc, or the like).
Macro definition of attributes is performed by the header file<ri_root>\include\os\option.h, which is called from the standard header file <ri_root>\include\kernel.h and the ITRON general definition header file <ri_root>\include\itron.h.

Table 15-3 Processing Program Attributes

Macro	Value	Description
TA_HLNG	0x0000	Start a processing unit through a C language interface.
TA_ASM	0x0001	Start a processing unit through an assembly language interface.
TA_ACT	0x0002	Task is activated after the creation.
TA_DISPREEMPT	0x4000	Preemption is disabled at task activation.
TA_ENAINT	0x0000	All interrupts are enabled at task activation.
TA_DISINT	0x8000	All interrupts are disabled at task activation.
TA_STA	0x0002	Cyclic handlers is in an operational state after the creation.
TA_PHS	0x0004	Cyclic handler is activated preserving the activation phase.

15.3.3 Management object attributes

The following lists the management object attributes acquired by issuing service calls (ref_sem, ref_flg, or the like).
Macro definition of attributes is performed by the standard header file <ri_root>\include\kernel.h, which is called from the header file<ri_root>\include\os\option.h and the ITRON general definition header file <ri_root>\include\itron.h.

Table 15-4 Management Object Attributes

Macro	Value	Description
TA_TFIFO	0x0000	Task wait queue is in FIFO order.
TA_TPRI	0x0001	Task wait queue is in task priority order.
TA_WSGL	0x0000	Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL	0x0002	Multiple tasks are allowed to be in the WAITING state for the eventflag.
TA_CLR	0x0004	Bit pattern is cleared when a task is released from the WAITING state for eventflag.
TA_MFIFO	0x0000	Message wait queue is in FIFO order.
TA_MPRI	0x0002	Message wait queue is in message priority order.

15.3.4 Service call operating modes

The following lists the service call operating modes used when issuing service calls (act_tsk, wup_tsk, or the like).

Macro definition of operating modes is performed by the header file<ri_root>\include\os\option.h, which is called from the standard header file <ri_root>\include\kernel.h and the ITRON general definition header file <ri_root>\include\itron.h.

Table 15-5 Service Call Operating Modes

Macro	Value	Description
TSK_SELF	0	Invoking task
TPRI_INI	0	Initial priority
TMO_FEVR	-1	Waiting forever
TMO_POL	0	Polling
TWF_ANDW	0x00	AND waiting condition
TWF_ORW	0x01	OR waiting condition
TPRI_SELF	0	Current priority of the Invoking task

15.3.5 Return value

The following lists the values returned from service calls.

Macros for the return values are defined in the header file <ri_root>\include\os\error.h and option.h, which are called from the standard header file <ri_root>\include\kernel.h and the common macro definition file for ITRON specifications <ri_root>\include\itron.h.

Table 15-6 Return Value

Macro	Value	Description
E_OK	0	Normal completion
E_NOSPT	-9	Unsupported function
E_RSFN	-10	Invalid function code
E_RSATR	-11	Invalid attribute
E_PAR	-17	Parameter error
E_ID	-18	Invalid ID number
E_CTX	-25	Context error.
E_ILUSE	-28	Illegal service call use
E_NOMEM	-33	Insufficient memory
E_OBJ	-41	Object state error
E_NOEXS	-42	Non-existent object
E_QOVR	-43	Queue overflow
E_RLWAI	-49	Forced release from the WAITING state
E_TMOUT	-50	Polling failure or timeout
FALSE	0	False
TRUE	1	True

15.3.6 Kernel configuration constants

The configuration constants are listed below.

The macro definitions of the configuration constants are made in the header file <ri_root>\include\os\component.h, which is called from <ri_root>\include\itron.h. Note, however, that some numerical values with variable macro definitions are defined in the system information header file, in accordance with the settings in the system configuration file.

Table 15-7 Priority Range

Macro	Value	Description
TMIN_TPRI	1	Minimum task priority
TMAX_TPRI	variable	Maximum task priority
TMIN_MPRI	1	Minimum message priority
TMAX_MPRI	0x7fff	Maximum message priority

Table 15-8 Version Information

Macro	Value	Description
TKERNEL_MAKER	0x011b	Kernel maker code
TKERNEL_PRID	0x0000	Identification number of kernel
TKERNEL_SPVER	0x5403	Version number of the ITRON Specification
TKERNEL_PRVER	0x01*xx*	Version number of the kernel

Table 15-9 Maximum Queuing Count

Macro	Value	Description
TMAX_ACTCNT	127	Maximum task activation request count
TMAX_WUPCNT	127	Maximum task wakeup request count
TMAX_SUSCNT	127	Maximum suspension count

Table 15-10 Number of Bits in Bit Patterns

Macro	Value	Description
TBIT_FLGPTN	32	Number of bits in the an eventflag

Table 15-11 Base Clock Interval

Macro	Value	Description
TIC_NUME	variable	base clock interval numerator
TIC_DENO	1	base clock interval denominator

15.4 Conditional Compile Macro

The header file of the RI850V4 is conditionally compiled by the following macros.

Define macros (such as the compiler activation option -D) according to the environment used when building the source files that include the header file of the RI850V4.

Table 15-12 Conditional Compile Macros

Classification	Macro	Description
C compiler package	__rel__	CC-RH is used. Add two underscores before and after "rel".
	__ghs__	CCV850 is used. Add two underscores before and after "ghs".
Coding language	__asm__	The assembly language is used. Add two underscores before and after "asm".

CHAPTER 16 SERVICE CALLS

This chapter describes the service calls supported by the RI850V4.

16.1 Outline

The service calls provided by the RI850V4 are service routines provided for indirectly manipulating the resources (tasks, semaphores, etc.) managed by the RI850V4 from a processing program.

The service calls provided by the RI850V4 are listed below by management module.

- Task management functions

  act_tsk, iact_tsk, can_act, ican_act, sta_tsk, ista_tsk, ext_tsk, ter_tsk, chg_pri, ichg_pri, get_pri, iget_pri, ref_tsk, iref_tsk, ref_tst, iref_tst

- Task dependent synchronization functions

  slp_tsk, tslp_tsk, wup_tsk, iwup_tsk, can_wup, ican_wup, rel_wai, irel_wai, sus_tsk, isus_tsk, rsm_tsk, irsm_tsk, frsm_tsk, ifrsm_tsk, dly_tsk

- Synchronization and communication functions (semaphores)

  wai_sem, pol_sem, ipol_sem, twai_sem, sig_sem, isig_sem, ref_sem, iref_sem

- Synchronization and communication functions (eventflags)

  set_flg, iset_flg, clr_flg, iclr_flg, wai_flg, pol_flg, ipol_flg, twai_flg, ref_flg, iref_flg

- Synchronization and communication functions (data queues)

  snd_dtq, psnd_dtq, ipsnd_dtq, tsnd_dtq, fsnd_dtq, ifsnd_dtq, rcv_dtq, prcv_dtq, iprcv_dtq, trcv_dtq, ref_dtq, iref_dtq

- Synchronization and communication functions (mailboxes)

  snd_mbx, isnd_mbx, rcv_mbx, prcv_mbx, iprcv_mbx, trcv_mbx, ref_mbx, iref_mbx

- Extended synchronization and communication functions (mutexes)

  loc_mtx, ploc_mtx, tloc_mtx, unl_mtx, ref_mtx, iref_mtx

- Memory pool management functions (fixed-sized memory pools)

  get_mpf, pget_mpf, ipget_mpf, tget_mpf, rel_mpf, irel_mpf, ref_mpf, iref_mpf

- Memory pool management functions (variable-sized memory pools)

  get_mpl, pget_mpl, ipget_mpl, tget_mpl, rel_mpl, irel_mpl, ref_mpl, iref_mpl

- Time management functions

  set_tim, iset_tim, get_tim, iget_tim, sta_cyc, ista_cyc, stp_cyc, istp_cyc, ref_cyc, iref_cyc

- System state management functions

  rot_rdq, irot_rdq, vsta_sch, get_tid, iget_tid, loc_cpu, iloc_cpu, unl_cpu, iunl_cpu, sns_loc, dis_dsp, ena_dsp, sns_dsp, sns_ctx, sns_dpn

- Service call management functions

  cal_svc, ical_svc

16.1.1 Call service call

The method for calling service calls from processing programs coded either in C or assembly language is described below.

- C language
  By calling using the same method as for normal C functions, service call parameters are handed over to the RI850V4 as arguments and the relevant processing is executed.

- Assembly language
  When issuing a service call from a processing program coded in assembly language, set parameters and the return address according to the calling rules prescribed in the C compiler used as the development environment and call the function using the jarl instruction; the service call parameters are then transferred to the RI850V4 as arguments and the relevant processing will be executed.

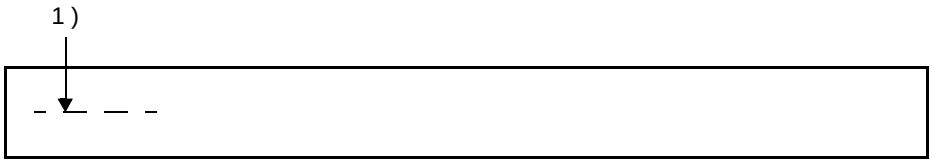
Note To call the service calls provided by the RI850V4 from a processing program, the header files listed below must be coded (include processing).

kernel.h: Standard header file

kernel_id.h: System information header file

16.2 Explanation of Service Call

The following explains the service calls supported by the RI850V4, in the format shown below.



2) → **Outline**

3) → **C format**

4) → **Parameter(s)**

I/O	Parameter	Description

5) → **Explanation**

6) → **Return value**

Macro	Value	Description

1) Name
Indicates the name of the service call.

2) Outline
Outlines the functions of the service call.

3) C format
Indicates the format to be used when describing a service call to be issued in C language.

4) Parameter(s)
Service call parameters are explained in the following format.

I/O	Parameter	Description
A	B	C

A) Parameter classification
I: Parameter input to RI850V4.
O: Parameter output from RI850V4.

B) Parameter data type

C) Description of parameter

5) Explanation
Explains the function of a service call.

6) Return value
Indicates a service call's return value using a macro and value.

Macro	Value	Description
A	B	C

A) Macro of return value

B) Value of return value

C) Description of return value

can_act
ican_act

Outline

Cancel task activation requests.

C format

```
ER_UINT can_act (ID tskid);
ER_UINT ican_act (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *tskid*;	ID number of the task for cancelling activation requests. TSK_SELF: Invoking task. Value: ID number of the task for cancelling activation requests.

Explanation

This service call cancels all of the activation requests queued to the task specified by parameter *tskid* (sets the activation request counter to 0x0).

When this service call is terminated normally, the number of cancelled activation requests is returned.

Note This service call does not perform status manipulation processing but performs the setting of activation request counter. Therefore, the task does not move from a state such as the READY state to the DORMANT state.

Return value

Macro	Value	Description
E_ID	-18	Invalid ID number. - *tskid* < 0x0 - *tskid* > Maximum ID number - When this service call was issued from a non-task, TSK_SELF was specified *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.
-	Positive value	Normal completion (activation request count).

16.2.1 Task management functions

The following shows the service calls provided by the RI850V4 as the task management functions.

Table 16-1 Task Management Functions

Service Call	Function	Origin of Service Call
act_tsk	Activate task (queues an activation request)	Task, Non-task, Initialization routine
iact_tsk	Activate task (queues an activation request)	Task, Non-task, Initialization routine
can_act	Cancel task activation requests	Task, Non-task, Initialization routine
ican_act	Cancel task activation requests	Task, Non-task, Initialization routine
sta_tsk	Activate task (does not queue an activation request)	Task, Non-task, Initialization routine
ista_tsk	Activate task (does not queue an activation request)	Task, Non-task, Initialization routine
ext_tsk	Terminate invoking task	Task
ter_tsk	Terminate task	Task, Initialization routine
chg_pri	Change task priority	Task, Non-task, Initialization routine
ichg_pri	Change task priority	Task, Non-task, Initialization routine
get_pri	Reference task priority	Task, Non-task, Initialization routine
iget_pri	Reference task priority	Task, Non-task, Initialization routine
ref_tsk	Reference task state	Task, Non-task, Initialization routine
iref_tsk	Reference task state	Task, Non-task, Initialization routine
ref_tst	Reference task state (simplified version)	Task, Non-task, Initialization routine
iref_tst	Reference task state (simplified version)	Task, Non-task, Initialization routine

act_tsk
iact_tsk

Outline

Activate task (queues an activation request).

C format

```
ER      act_tsk (ID tskid);
ER      iact_tsk (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	ID tskid;	ID number of the task to be activated. TSK_SELF: Invoking task. Value: ID number of the task to be activated.

Explanation

These service calls move a task specified by parameter *tskid* from the DORMANT state to the READY state.

As a result, the target task is queued at the end on the ready queue corresponding to the initial priority and becomes subject to scheduling by the RI850V4.

If the target task has been moved to a state other than the DORMANT state when this service call is issued, this service call does not move the state but increments the activation request counter (by added 0x1 to the wakeup request counter).

Note 1 The activation request counter managed by the RI850V4 is configured in 7-bit widths. If the number of activation requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

Note 2 Extended information specified in Task information is passed to the task activated by issuing these service calls.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0x0 - *tskid* > Maximum ID number - When this service call was issued from a non-task, TSK_SELF was specified *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified task is not registered.
E_QOVR	-43	Queue overflow. - Activation request count exceeded 127.

sta_tsk
ista_tsk

Outline

Activate task (does not queue an activation request).

C format

```
ER      sta_tsk (ID tskid, VP_INT stacd);
ER      ista_tsk (ID tskid, VP_INT stacd);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task to be activated.
I	`VP_INT stacd;`	Start code (extended information) of the task.

Explanation

These service calls move a task specified by parameter *tskid* from the DORMANT state to the READY state.

As a result, the target task is queued at the end on the ready queue corresponding to the initial priority and becomes subject to scheduling by the RI850V4.

This service call does not perform queuing of activation requests. If the target task is in a state other than the DORMANT state, the status manipulation processing for the target task is therefore not performed but "E_OBJ" is returned

Specify for parameter *stacd* the extended information transferred to the target task.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* ≤ 0x0 - *tskid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_OBJ	-41	Object state error - Specified task is not in the DORMANT state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.

ext_tsk

Outline

Terminate invoking task.

C format

```
void    ext_tsk (void);
```

Parameter(s)

None.

Explanation

This service call moves an invoking task from the RUNNING state to the DORMANT state.
As a result, the invoking task is unlinked from the ready queue and excluded from the RI850V4 scheduling subject.
If an activation request has been queued to the invoking task (the activation request counter is not set to 0x0) when this service call is issued, this service call moves the task from the RUNNING state to the DORMANT state, decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter), and then moves the task from the DORMANT state to the READY state.

Note 1 When moving a task from the RUNNING state to the DORMANT state, this service call initializes the following information to values that are set during task creation.

- Current priority
- Wakeup request count
- Suspension count
- interrupt state

If an invoking task has locked a mutex, the locked state is released at the same time (processing equivalent to unl_mtx).

Note 2 When the return instruction is issued in a task, the same processing as ext_tsk is performed.

Return value

None.

ter_tsk

Outline

Terminate task.

C format

```
ER      ter_tsk (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *tskid*;	ID number of the task to be terminated.

Explanation

This service call forcibly moves a task specified by parameter *tskid* to the DORMANT state.
As a result, the target task is excluded from the RI850V4 scheduling subject.
If an activation request has been queued to the target task (the activation request counter is not set to 0x0) when this service call is issued, this service call moves the task to the DORMANT state, decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter), and then moves the task from the DORMANT state to the READY state.

Note When moving a task to the DORMANT state, this service call initializes the following information to values that are set during task creation.

- Current priority
- Wakeup request count
- Suspension count
- Interrupt state

If the target task has locked a mutex, the locked state is released at the same time (processing equivalent to unl_mtx).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* ≤ 0x0 - *tskid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_ILUSE	-28	Illegal service call use. - Specified task is an invoking task.
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.

chg_pri
ichg_pri

Outline

Change task priority.

C format

```
ER      chg_pri (ID tskid, PRI tskpri);
ER      ichg_pri (ID tskid, PRI tskpri);
```

Parameter(s)

I/O	Parameter	Description
I	ID tskid;	ID number of the task whose priority is to be changed. TSK_SELF: Invoking task. Value: ID number of the task whose priority is to be changed.
I	PRI tskpri;	New base priority of the task. TPRI_INI: Initial priority. Value: New base priority.

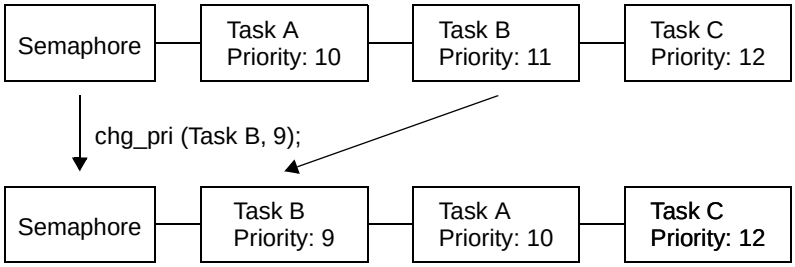
Explanation

These service calls change the priority of the task specified by parameter *tskid* (current priority) to a value specified by parameter *tskpri*.

If the target task is in the RUNNING or READY state after this service call is issued, this service call re-queues the task at the end of the ready queue corresponding to the priority specified by parameter *tskpri*, following priority change processing.

Note When the target task is queued to a wait queue in the order of priority, the wait order may change due to issue of this service call.

Example When three tasks (task A: priority level 10, task B: priority level 11, task C: priority level 12) are queued to the semaphore wait queue in the order of priority, and the priority level of task B is changed from 11 to 9, the wait order will be changed as follows.



Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tskpri* < 0x0 - *tskpri* > Maximum priority
E_ID	-18	Invalid ID number. - *tskid* < 0x0 - *tskid* > Maximum ID number - When this service call was issued from a non-task, TSK_SELF was specified *tskid*.
E_CTX	-25	Context error. - This service call was issued int the CPU locked state.
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.

get_pri
iget_pri

Outline

Reference task priority.

C format

```
ER      get_pri (ID tskid, PRI *p_tskpri);
ER      iget_pri (ID tskid, PRI *p_tskpri);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task to reference. TSK_SELF: Invoking task. Value: ID number of the task to reference.
O	`PRI *p_tskpri;`	Current priority of specified task.

Explanation

Stores current priority of the task specified by parameter *tskid* in the area specified by parameter *p_tskpri*.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0x0 - *tskid* > Maximum ID number - When this service call was issued from a non-task, TSK_SELF was specified *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.

ref_tsk
iref_tsk

Outline

Reference task state.

C format

```
ER      ref_tsk (ID tskid, T_RTSK *pk_rtsk);
ER      iref_tsk (ID tskid, T_RTSK *pk_rtsk);
```

Parameter(s)

I/O	Parameter	Description
I	ID tskid;	ID number of the task to referenced. TSK_SELF: Invoking task. Value: ID number of the task to referenced.
O	T_RTSK *pk_rtsk;	Pointer to the packet returning the task state.

[Task state packet: T_RTSK]

```
typedef struct  t_rtsk {
    STAT    tskstat;        /*Current state*/
    PRI     tskpri;         /*Current priority*/
    PRI     tskbpri;        /*Reserved for future use*/
    STAT    tskwait;        /*Reason for waiting*/
    ID      wobjid;         /*Object ID number for which the task is waiting*/
    TMO     lefttmo;        /*Remaining time until timeout*/
    UINT    actcnt;         /*Activation request count*/
    UINT    wupcnt;         /*Wakeup request count*/
    UINT    suscnt;         /*Suspension count*/
    ATR     tskatr;         /*Attribute*/
    PRI     itskpri;        /*Initial priority*/
    ID      memid;          /*Reserved for future use*/
} T_RTSK;
```

Explanation

Stores task state packet (current state, current priority, etc.) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtsk*.

Note For details about the task state packet, refer to "15.2.1 Task state packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0x0 - *tskid* > Maximum ID number - When this service call was issued from a non-task, TSK_SELF was specified *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-Existent object. - Specified task is not registered.

ref_tst
iref_tst

Outline

Reference task state (simplified version).

C format

```
ER      ref_tst (ID tskid, T_RTST *pk_rtst);
ER      iref_tst (ID tskid, T_RTST *pk_rtst);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task to be referenced. TSK_SELF: Invoking task. Value: ID number of the task to be referenced.
O	`T_RTST *pk_rtst;`	Pointer to the packet returning the task state.

[Task state packet (simplified version): T_RTST]

```
typedef struct  t_rtst {
    STAT    tskstat;        /*Current state*/
    STAT    tskwait;        /*Reason for waiting*/
} T_RTST;
```

Explanation

Stores task state packet (current state, reason for waiting) of the task specified by parameter *tskid* in the area specified by parameter *pk_rtst*.

Used for referencing only the current state and reason for wait among task information.

Response becomes faster than using ref_tsk or iref_tsk because only a few information items are acquired.

Note For details about the task state packet (simplified version), refer to "15.2.2 Task state packet (simplified version)".

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *tskid* < 0x0 - *tskid* > Maximum ID number - When this service call was issued from a non-task, TSK_SELF was specified *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.

16.2.2 Task dependent synchronization functions

The following shows the service calls provided by the RI850V4 as the task dependent synchronization functions.

Table 16-2 Task Dependent Synchronization Functions

Service Call	Function	Origin of Service Call
slp_tsk	Put task to sleep (waiting forever)	Task
tslp_tsk	Put task to sleep (with timeout)	Task
wup_tsk	Wakeup task	Task, Non-task, Initialization routine
iwup_tsk	Wakeup task	Task, Non-task, Initialization routine
can_wup	Cancel task wakeup requests	Task, Non-task, Initialization routine
ican_wup	Cancel task wakeup requests	Task, Non-task, Initialization routine
rel_wai	Release task from waiting	Task, Non-task, Initialization routine
irel_wai	Release task from waiting	Task, Non-task, Initialization routine
sus_tsk	Suspend task	Task, Non-task, Initialization routine
isus_tsk	Suspend task	Task, Non-task, Initialization routine
rsm_tsk	Resume suspended task	Task, Non-task, Initialization routine
irsm_tsk	Resume suspended task	Task, Non-task, Initialization routine
frsm_tsk	Forcibly resume suspended task	Task, Non-task, Initialization routine
ifrsm_tsk	Forcibly resume suspended task	Task, Non-task, Initialization routine
dly_tsk	Delay task	Task

slp_tsk

Outline

Put task to sleep (waiting forever).

C format

```
ER      slp_tsk (void);
```

Parameter(s)

None.

Explanation

As a result, the invoking task is unlinked from the ready queue and excluded from the RI850V4 scheduling subject.

If a wakeup request has been queued to the target task (the wakeup request counter is not set to 0x0) when this service call is issued, this service call does not move the state but decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter).

The sleeping state is cancelled in the following cases, and then moved to the READY state.

Sleeping State Cancel Operation	Return Value
A wakeup request was issued as a result of issuing wup_tsk.	E_OK
A wakeup request was issued as a result of issuing iwup_tsk.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

tslp_tsk

Outline

Put task to sleep (with timeout).

C format

```
ER      tslp_tsk (TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	TMO *tmout*;	Specified timeout (unit:millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified timeout.

Explanation

This service call moves an invoking task from the RUNNING state to the WAITING state (sleeping state).
As a result, the invoking task is unlinked from the ready queue and excluded from the RI850V4 scheduling subject.
If a wakeup request has been queued to the target task (the wakeup request counter is not set to 0x0) when this service call is issued, this service call does not move the state but decrements the wakeup request counter (by subtracting 0x1 from the wakeup request counter).
The sleeping state is cancelled in the following cases, and then moved to the READY state.

Sleeping State Cancel Operation	Return Value
A wakeup request was issued as a result of issuing wup_tsk.	E_OK
A wakeup request was issued as a result of issuing iwup_tsk.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

Note When TMO_FEVR is specified for wait time *tmout*, processing equivalent to slp_tsk will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tmout* < TMO_FEVR

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Timeout. - Polling failure or timeout.

wup_tsk
iwup_tsk

Outline

Wakeup task.

C format

```
ER      wup_tsk (ID tskid);
ER      iwup_tsk (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	ID tskid;	ID number of the task to be woken up. TSK_SELF: Invoking task. Value: ID number of the task to be woken up.

Explanation

These service calls cancel the WAITING state (sleeping state) of the task specified by parameter *tskid*.

As a result, the target task is moved from the sleeping state to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

If the target task is in a state other than the sleeping state when this service call is issued, this service call does not move the state but increments the wakeup request counter (by added 0x1 to the wakeup request counter).

Note The wakeup request counter managed by the RI850V4 is configured in 7-bit widths. If the number of wakeup requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0x0 - *tskid* > Maximum ID number - When this service call was issued from a non-task, TSK_SELF was specified *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified task is not registered.
E_QOVR	-43	Queue overflow. - Wakeup request count exceeded 127.

can_wup
ican_wup

Outline

Cancel task wakeup requests.

C format

```
ER_UINT can_wup (ID tskid);
ER_UINT ican_wup (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *tskid*;	ID number of the task for cancelling wakeup requests. TSK_SELF: Invoking task. Value: ID number of the task for cancelling wakeup requests.

Explanation

These service calls cancel all of the wakeup requests queued to the task specified by parameter *tskid* (the wakeup request counter is set to 0x0).

When this service call is terminated normally, the number of cancelled wakeup requests is returned.

Return value

Macro	Value	Description
E_ID	-18	Invalid ID number. - *tskid* < 0x0 - *tskid* > Maximum ID number - When this service call was issued from a non-task, TSK_SELF was specified *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.
other	-	Normal completion (wakeup request count).

rel_wai
irel_wai

Outline

Release task from waiting.

C format

```
ER      rel_wai (ID tskid);
ER      irel_wai (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task to be released from waiting.

Explanation

These service calls forcibly cancel the WAITING state of the task specified by parameter *tskid*.

As a result, the target task unlinked from the wait queue and is moved from the WAITING state to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

"E_RLWAI" is returned from the service call that triggered the move to the WAITING state (slp_tsk, wai_sem, or the like) to the task whose WAITING state is cancelled by this service call.

Note 1 This service call does not perform queuing of forced cancellation requests. If the target task is in a state other than the WAITING or WAITING-SUSPENDED state, "E_OBJ" is returned.

Note 2 The SUSPENDED state is not cancelled by these service calls.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* ≤ 0x0 - *tskid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_OBJ	-41	Object state error. - Specified task is neither in the WAITING state nor WAITING-SUSPENDED state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.

sus_tsk
isus_tsk

Outline

Suspend task.

C format

```
ER      sus_tsk (ID tskid);
ER      isus_tsk (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *tskid*;	ID number of the task to be suspended. TSK_SELF: Invoking task. Value: ID number of the task to be suspended.

Explanation

These service calls add 0x1 to the suspend request counter for the task specified by parameter *tskid*, and then move the target task from the RUNNING state to the SUSPENDED state, from the READY state to the SUSPENDED state, or from the WAITING state to the WAITING-SUSPENDED state.

If the target task has moved to the SUSPENDED or WAITING-SUSPENDED state when this service call is issued, the counter manipulation processing is not performed but only the suspend request counter increment processing is executed.

Note The suspend request counter managed by the RI850V4 is configured in 7-bit widths. If the number of suspend requests exceeds the maximum count value 127 as a result of issuing this service call, the counter manipulation processing is therefore not performed but "E_QOVR" is returned.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* < 0x0 - *tskid* > Maximum ID number - When this service call was issued from a non-task, TSK_SELF was specified *tskid*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state. - When this service call was issued in the dispatching disabled state, invoking task was specified *tskid*.

Macro	Value	Description
E_OBJ	-41	Object state error. - Specified task is in the DORMANT state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.
E_QOVR	-43	Queue overflow. - Suspension count exceeded 127.

rsm_tsk
irsm_tsk

Outline

Resume suspended task.

C format

```
ER      rsm_tsk (ID tskid);
ER      irsm_tsk (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *tskid*;	ID number of the task to be resumed.

Explanation

This service call subtracts 0x1 from the suspend request counter for the task specified by parameter *tskid*, and then cancels the SUSPENDED state of the target task.

As a result, the target task is moved from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.

If a suspend request is queued (subtraction result is other than 0x0) when this service call is issued, the counter manipulation processing is not performed but only the suspend request counter decrement processing is executed.

Note This service call does not perform queuing of cancellation requests. If the target task is in a state other than the SUSPENDED or WAITING-SUSPENDED state, "E_OBJ" is therefore returned.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* ≤ 0x0 - *tskid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_OBJ	-41	Object state error. - Specified task is neither in the SUSPENDED state nor WAITING-SUSPENDED state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.

frsm_tsk
ifrsm_tsk

Outline

Forcibly resume suspended task.

C format

```
ER      frsm_tsk (ID tskid);
ER      ifrsm_tsk (ID tskid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID tskid;`	ID number of the task to be resumed.

Explanation

These service calls cancel all of the suspend requests issued for the task specified by parameter *tskid* (by setting the suspend request counter to 0x0). As a result, the target task moves from the SUSPENDED state to the READY state, or from the WAITING-SUSPENDED state to the WAITING state.

Note This service call does not perform queuing of cancellation requests. If the target task is in a state other than the SUSPENDED or WAITING-SUSPENDED state, "E_OBJ" is therefore returned.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *tskid* ≤ 0x0 - *tskid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_OBJ	-41	Object state error. - Specified task is neither in the SUSPENDED state nor WAITING-SUSPENDED state.
E_NOEXS	-42	Non-existent object. - Specified task is not registered.

dly_tsk

Outline

Delay task.

C format

```
ER      dly_tsk (RELTIM dlytim);
```

Parameter(s)

I/O	Parameter	Description
I	`RELTIM dlytim;`	Amount of time to delay the invoking task (unit:millisecond).

Explanation

This service call moves the invoking task from the RUNNING state to the WAITING state (delayed state).
As a result, the invoking task is unlinked from the ready queue and excluded from the RI850V4 scheduling subject.
The delayed state is cancelled in the following cases, and then moved to the READY state.

Delayed State Cancel Operation	Return Value
Delay time specified by parameter *dlytim* has elapsed.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

16.2.3 Synchronization and communication functions (semaphores)

The following shows the service calls provided by the RI850V4 as the synchronization and communication functions (semaphores).

Table 16-3 Synchronization and Communication Functions (Semaphores)

Service Call	Function	Origin of Service Call
wai_sem	Acquire semaphore resource (waiting forever)	Task
pol_sem	Acquire semaphore resource (polling)	Task, Non-task, Initialization routine
ipol_sem	Acquire semaphore resource (polling)	Task, Non-task, Initialization routine
twai_sem	Acquire semaphore resource (with timeout)	Task
sig_sem	Release semaphore resource	Task, Non-task, Initialization routine
isig_sem	Release semaphore resource	Task, Non-task, Initialization routine
ref_sem	Reference semaphore state	Task, Non-task, Initialization routine
iref_sem	Reference semaphore state	Task, Non-task, Initialization routine

wai_sem

Outline

Acquire semaphore resource (waiting forever).

C format

```
ER      wai_sem (ID semid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *semid*;	ID number of the semaphore from which resource is acquired.

Explanation

This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).

If no resources are acquired from the target semaphore when this service call is issued (no available resources exist), this service call does not acquire resources but queues the invoking task to the target semaphore wait queue and moves it from the RUNNING state to the WAITING state (resource acquisition wait state).

The WAITING state for a semaphore resource is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Semaphore Resource Cancel Operation	Return Value
The resource was returned to the target semaphore as a result of issuing sig_sem.	E_OK
The resource was returned to the target semaphore as a result of issuing isig_sem.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

Note Invoking tasks are queued to the target semaphore wait queue in the order defined during configuration (FIFO order or priority order).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *semid* ≤ 0x0 - *semid* > Maximum ID number

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified semaphore is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

pol_sem
ipol_sem

Outline

Acquire semaphore resource (polling).

C format

```
ER      pol_sem (ID semid);
ER      isem_sem (ID semid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID semid;`	ID number of the semaphore from which resource is acquired.

Explanation

This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).

If a resource could not be acquired from the target semaphore (semaphore counter is set to 0x0) when this service call is issued, the counter manipulation processing is not performed but "E_TMOUT" is returned.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *semid* ≤ 0x0 - *semid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified semaphore is not registered.
E_TMOUT	-50	Polling failure. - The resource counter of the target semaphore is 0x0.

twai_sem

Outline

Acquire semaphore resource (with timeout).

C format

```
ER      twai_sem (ID semid, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	ID *semid*;	ID number of the semaphore from which resource is acquired.
I	TMO *tmout*;	Specified timeout (unit:millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified timeout.

Explanation

This service call acquires a resource from the semaphore specified by parameter *semid* (subtracts 0x1 from the semaphore counter).

If no resources are acquired from the target semaphore when service call is issued this (no available resources exist), this service call does not acquire resources but queues the invoking task to the target semaphore wait queue and moves it from the RUNNING state to the WAITING state with timeout (resource acquisition wait state).

The WAITING state for a semaphore resource is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Semaphore Resource Cancel Operation	Return Value
The resource was returned to the target semaphore as a result of issuing sig_sem.	E_OK
The resource was returned to the target semaphore as a result of issuing isig_sem.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

Note 1 Invoking tasks are queued to the target semaphore wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 TMO_FEVR is specified for wait time *tmout*, processing equivalent to wai_sem will be executed. When TMO_POL is specified, processing equivalent to pol_sem /ipol_sem will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_PAR	-17	Parameter error. - *tmout* < TMO_FEVR
E_ID	-18	Invalid ID number. - *semid* ≤ 0x0 - *semid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified semaphore is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Timeout. - Polling failure or timeout.

sig_sem
isig_sem

Outline

Release semaphore resource.

C format

```
ER      sig_sem (ID semid);
ER      isig_sem (ID semid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID semid;`	ID number of the semaphore to which resource is released.

Explanation

These service calls return the resource to the semaphore specified by parameter *semid* (adds 0x1 to the semaphore counter).

If a task is queued in the wait queue of the target semaphore when this service call is issued, the counter manipulation processing is not performed but the resource is passed to the relevant task (first task of wait queue).

As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (WAITING state for a semaphore resource) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note With the RI850V4, the maximum possible number of semaphore resources (maximum resource count) is defined during configuration. If the number of resources exceeds the specified maximum resource count, this service call therefore does not return the acquired resources (addition to the semaphore counter value) but returns E_QOVR.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *semid* ≤ 0x0 - *semid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified semaphore is not registered.
E_QOVR	-43	Queue overflow. - Resource count exceeded maximum resource count.

ref_sem
iref_sem

Outline

Reference semaphore state.

C format

```
ER      ref_sem (ID semid, T_RSEM *pk_rsem);
ER      iref_sem (ID semid, T_RSEM *pk_rsem);
```

Parameter(s)

I/O	Parameter	Description
I	`ID semid;`	ID number of the semaphore to be referenced.
O	`T_RSEM *pk_rsem;`	Pointer to the packet returning the semaphore state.

[Semaphore state packet: T_RSEM]

```
typedef struct  t_rsem {
    ID      wtskid;         /*Existence of waiting task*/
    UINT    semcnt;         /*Current resource count*/
    ATR     sematr;         /*Attribute*/
    UINT    maxsem;         /*Maximum resource count*/
} T_RSEM;
```

Explanation

Stores semaphore state packet (ID number of the task at the head of the wait queue, current resource count, etc.) of the semaphore specified by parameter *semid* in the area specified by parameter *pk_rsem*.

Note For details about the semaphore state packet, refer to "15.2.3 Semaphore state packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *semid* ≤ 0x0 - *semid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified semaphore is not registered.

16.2.4 Synchronization and communication functions (eventflags)

The following shows the service calls provided by the RI850V4 as the synchronization and communication functions (eventflags).

Table 16-4 Synchronization and Communication Functions (Eventflags)

Service Call	Function	Origin of Service Call
set_flg	Set eventflag	Task, Non-task, Initialization routine
iset_flg	Set eventflag	Task, Non-task, Initialization routine
clr_flg	Clear eventflag	Task, Non-task, Initialization routine
iclr_flg	Clear eventflag	Task, Non-task, Initialization routine
wai_flg	Wait for eventflag (waiting forever)	Task
pol_flg	Wait for eventflag (polling)	Task, Non-task, Initialization routine
ipol_flg	Wait for eventflag (polling)	Task, Non-task, Initialization routine
twai_flg	Wait for eventflag (with timeout)	Task
ref_flg	Reference eventflag state	Task, Non-task, Initialization routine
iref_flg	Reference eventflag state	Task, Non-task, Initialization routine

set_flg
iset_flg

Outline

Set eventflag.

C format

```
ER      set_flg (ID flgid, FLGPTN setptn);
ER      iset_flg (ID flgid, FLGPTN setptn);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag to be set.
I	`FLGPTN setptn;`	Bit pattern to set.

Explanation

These service calls set the result of logical OR operating the bit pattern of the eventflag specified by parameter *flgid* and the bit pattern specified by parameter *setptn* as the bit pattern of the target eventflag.

If the required condition of the task queued to the target eventflag wait queue is satisfied when this service call is issued, the relevant task is unlinked from the wait queue at the same time as bit pattern setting processing.

As a result, the relevant task is moved from the WAITING state (WAITING state for an eventflag) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 If the bit pattern set to the target eventflag is B'1100 and the bit pattern specified by parameter *setptn* is B'1010 when this service call is issued, the bit pattern of the target eventflag is set to B'1110.

Note 2 When the TA_WMUL attribute is specified for the target eventflag, the range of tasks to be checked on "whether issuing of this service call satisfies the required condition" differs depending on whether the TA_CLR attribute is also specified.

- When TA_CLR is specified
  Check begins from the task at the head of the wait queue and stops at the first task that meets the requirements.
- When TA_CLR is not specified
  All tasks placed in the wait queue are checked.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *flgid* ≤ 0x0 - *flgid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified eventflag is not registered.

clr_flg
iclr_flg

Outline

Clear eventflag.

C format

```
ER      clr_flg (ID flgid, FLGPTN clrptn);
ER      iclr_flg (ID flgid, FLGPTN clrptn);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag to be cleared.
I	`FLGPTN clrptn;`	Bit pattern to clear.

Explanation

This service call sets the result of logical AND operating the bit pattern set to the eventflag specified by parameter *flgid* and the bit pattern specified by parameter *clrptn* as the bit pattern of the target eventflag.

Note If the bit pattern set to the target eventflag is B'1100 and the bit pattern specified by parameter *clrptn* is B'1010 when this service call is issued, the bit pattern of the target eventflag is set to B'1000.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *flgid* ≤ 0x0 - *flgid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified eventflag is not registered.

wai_flg

Outline

Wait for eventflag (waiting forever).

C format

```
ER      wai_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag to wait for.
I	`FLGPTN waiptn;`	Wait bit pattern.
I	`MODE wfmode;`	Wait mode. TWF_ANDW: AND waiting condition. TWF_ORW: OR waiting condition.
O	`FLGPTN *p_flgptn;`	Bit pattern causing a task to be released from waiting.

Explanation

This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.

If a bit pattern that satisfies the required condition has been set for the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.

If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, the invoking task is queued to the target eventflag wait queue.

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (WAITING state for an eventflag).

The WAITING state for an eventflag is cancelled in the following cases, and then moved to the READY state.

WAITING State for an Eventflag Cancel Operation	Return Value
A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing set_flg.	E_OK
A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing iset_flg.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.

- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

Note 1 With the RI850V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TW_WSGL attribute) to which a wait task is queued, therefore, "E_ILUSE" is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL: Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL: Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 Invoking tasks are queued to the target event flag (TA_WMUL attribute) wait queue in the order defined during configuration (FIFO order or priority order).

Note 3 The RI850V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 4 If the WAITING state for an eventflag is forcibly released by issuing rel_wai or irel_wai, the contents of the area specified by parameter *p_flgptn* will be undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *waiptn* = 0x0 - *wfmode* is invalid.
E_ID	-18	Invalid ID number. - *flgid* ≤ 0x0 - *flgid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_ILUSE	-28	Illegal service call use. - There is already a task waiting for an eventflag with the TA_WSGL attribute.
E_NOEXS	-42	Non-existent object. - Specified eventflag is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

pol_flg
ipol_flg

Outline

Wait for eventflag (polling).

C format

```
ER      pol_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
ER      ipol_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag to wait for.
I	`FLGPTN waiptn;`	Wait bit pattern.
I	`MODE wfmode;`	Wait mode. TWF_ANDW: AND waiting condition. TWF_ORW: OR waiting condition.
O	`FLGPTN *p_flgptn;`	Bit pattern causing a task to be released from waiting.

Explanation

This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.

If the bit pattern that satisfies the required condition has been set to the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.

If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, "E_TMOUT" is returned.

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.

- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

Note 1 With the RI850V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TW_WSGL attribute) to which a wait task is queued, therefore, "E_ILUSE" is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL: Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL: Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 The RI850V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 3 If the bit pattern of the target event flag does not satisfy the required condition when this service call is issued, the contents in the area specified by parameter *p_flgptn* become undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *waiptn* = 0x0 - *wfmode* is invalid.
E_ID	-18	Invalid ID number. - *flgid* ≤ 0x0 - *flgid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_ILUSE	-28	Illegal service call use. - There is already a task waiting for an eventflag with the TA_WSGL attribute.
E_NOEXS	-42	Non-existent object. - Specified eventflag is not registered.
E_TMOUT	-50	Polling failure. - The bit pattern of the target eventflag does not satisfy the wait condition.

twai_flg

Outline

Wait for eventflag (with timeout).

C format

```
ER      twai_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag to wait for.
I	`FLGPTN waiptn;`	Wait bit pattern.
I	`MODE wfmode;`	Wait mode. TWF_ANDW: AND waiting condition. TWF_ORW: OR waiting condition.
O	`FLGPTN *p_flgptn;`	Bit pattern causing a task to be released from waiting.
I	`TMO tmout;`	Specified timeout (unit:millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified timeout.

Explanation

This service call checks whether the bit pattern specified by parameter *waiptn* and the bit pattern that satisfies the required condition specified by parameter *wfmode* are set to the eventflag specified by parameter *flgid*.

If a bit pattern that satisfies the required condition has been set for the target eventflag, the bit pattern of the target eventflag is stored in the area specified by parameter *p_flgptn*.

If the bit pattern of the target eventflag does not satisfy the required condition when this service call is issued, the invoking task is queued to the target eventflag wait queue.

As a result, the invoking task is unlinked from the ready queue and is moved from the RUNNING state to the WAITING state (WAITING state for an eventflag).

The WAITING state for an eventflag is cancelled in the following cases, and then moved to the READY state.

WAITING State for an Eventflag Cancel Operation	Return Value
A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing set_flg.	E_OK
A bit pattern that satisfies the required condition was set to the target eventflag as a result of issuing iset_flg.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

The following shows the specification format of required condition *wfmode*.

- *wfmode* = TWF_ANDW
  Checks whether all of the bits to which 1 is set by parameter *waiptn* are set as the target eventflag.

- *wfmode* = TWF_ORW
  Checks which bit, among bits to which 1 is set by parameter *waiptn*, is set as the target eventflag.

Note 1 With the RI850V4, whether to enable queuing of multiple tasks to the event flag wait queue is defined during configuration. If this service call is issued for the event flag (TW_WSGL attribute) to which a wait task is queued, therefore, "E_ILUSE" is returned regardless of whether the required condition is immediately satisfied.

TA_WSGL: Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL: Multiple tasks are allowed to be in the WAITING state for the eventflag.

Note 2 Invoking tasks are queued to the target event flag (TA_WMUL attribute) wait queue in the order defined during configuration (FIFO order or priority order).

Note 3 The RI850V4 performs bit pattern clear processing (0x0 setting) when the required condition of the target eventflag (TA_CLR attribute) is satisfied.

Note 4 If the event flag wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_flgptn* become undefined.

Note 5 TMO_FEVR is specified for wait time *tmout*, processing equivalent to wai_flg will be executed. When TMO_POL is specified, processing equivalent to pol_flg /ipol_flg will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *waiptn* = 0x0 - *wfmode* is invalid. - *tmout* < TMO_FEVR
E_ID	-18	Invalid ID number. - *flgid* ≤ 0x0 - *flgid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_ILUSE	-28	Illegal service call use. - There is already a task waiting for an eventflag with the TA_WSGL attribute.
E_NOEXS	-42	Non-existent object. - Specified eventflag is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Timeout. - Polling failure or timeout.

ref_flg
iref_flg

Outline

Reference eventflag state.

C format

```
ER      ref_flg (ID flgid, T_RFLG *pk_rflg);
ER      iref_flg (ID flgid, T_RFLG *pk_rflg);
```

Parameter(s)

I/O	Parameter	Description
I	`ID flgid;`	ID number of the eventflag to be referenced.
O	`T_RFLG *pk_rflg;`	Pointer to the packet returning the eventflag state.

[Eventflag state packet: T_RFLG]

```
typedef struct  t_rflg {
    ID      wtskid;         /*Existence of waiting task*/
    FLGPTN  flgptn;         /*Current bit pattern*/
    ATR     flgatr;         /*Attribute*/
} T_RFLG;
```

Explanation

Stores eventflag state packet (ID number of the task at the head of the wait queue, current bit pattern, etc.) of the eventflag specified by parameter *flgid* in the area specified by parameter *pk_rflg*.

Note For details about the eventflag state packet, refer to "15.2.4 Eventflag state packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *flgid* ≤ 0x0 - *flgid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified eventflag is not registered.

16.2.5 Synchronization and communication functions (data queues)

The following shows the service calls provided by the RI850V4 as the synchronization and communication functions (data queues).

Table 16-5 Synchronization and Communication Functions (Data Queues)

Service Call	Function	Origin of Service Call
snd_dtq	Send to data queue (waiting forever)	Task
psnd_dtq	Send to data queue (polling)	Task, Non-task, Initialization routine
ipsnd_dtq	Send to data queue (polling)	Task, Non-task, Initialization routine
tsnd_dtq	Send to data queue (with timeout)	Task
fsnd_dtq	Forced send to data queue	Task, Non-task, Initialization routine
ifsnd_dtq	Forced send to data queue	Task, Non-task, Initialization routine
rcv_dtq	Receive from data queue (waiting forever)	Task
prcv_dtq	Receive from data queue (polling)	Task, Non-task, Initialization routine
iprcv_dtq	Receive from data queue (polling)	Task, Non-task, Initialization routine
trcv_dtq	Receive from data queue (with timeout)	Task
ref_dtq	Reference data queue state	Task, Non-task, Initialization routine
iref_dtq	Reference data queue state	Task, Non-task, Initialization routine

snd_dtq

Outline

Send to data queue (waiting forever).

C format

```
ER      snd_dtq (ID dtqid, VP_INT data);
```

Parameter(s)

I/O	Parameter	Description
I	ID *dtqid*;	ID number of the data queue to which the data element is sent.
I	VP_INT *data*;	Data element to be sent to the data queue.

Explanation

This service call writes data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

If there is no available space for writing data in the data queue area of the target data queue when this service call is issued, this service call does not write data but queues the invoking task to the transmission wait queue of the target data queue and moves it from the RUNNING state to the WAITING state (data transmission wait state).

The sending WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

Sending WAITING State for a Data Queue Cancel Operation	Return Value
Available space was secured in the data queue area of the target data queue as a result of issuing rcv_dtq.	E_OK
Available space was secured in the data queue area of the target data queue as a result of issuing prcv_dtq.	E_OK
Available space was secured in the data queue area of the target data queue as a result of issuing iprcv_dtq.	E_OK
Available space was secured in the data queue area of the target data queue as a result of issuing trcv_dtq.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 Data is written to the data queue area of the target data queue in the order of the data transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target data queue in the order defined during configuration (FIFO order or priority order).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0x0 - *dtqid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified data queue is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

psnd_dtq
ipsnd_dtq

Outline

Send to data queue (polling).

C format

```
ER      psnd_dtq (ID dtqid, VP_INT data);
ER      ipsnd_dtq (ID dtqid, VP_INT data);
```

Parameter(s)

I/O	Parameter	Description
I	ID *dtqid*;	ID number of the data queue to which the data element is sent.
I	VP_INT *data*;	Data element to be sent to the data queue.

Explanation

These service calls write data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

If there is no available space for writing data in the data queue area of the target data queue when either of these service calls is issued, data is not written but E_TMOUT is returned.

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note Data is written to the data queue area of the target data queue in the order of the data transmission request.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0x0 - *dtqid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified data queue is not registered.
E_TMOUT	-50	Polling failure. - There is no space in the target data queue.

tsnd_dtq

Outline

Send to data queue (with timeout).

C format

```
ER      tsnd_dtq (ID dtqid, VP_INT data, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	ID *dtqid*;	ID number of the data queue to which the data element is sent.
I	VP_INT *data*;	Data element to be sent to the data queue.
I	TMO *tmout*;	Specified timeout (unit:millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified timeout.

Explanation

This service call writes data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

If there is no available space for writing data in the data queue area of the target data queue when this service call is issued, the service call does not write data but queues the invoking task to the transmission wait queue of the target data queue and moves it from the RUNNING state to the WAITING state with time (data transmission wait state).

The sending WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

Sending WAITING State for a Data Queue Cancel Operation	Return Value
An available space was secured in the data queue area of the target data queue as a result of issuing rcv_dtq.	E_OK
An available space was secured in the data queue area of the target data queue as a result of issuing prcv_dtq.	E_OK
An available space was secured in the data queue area of the target data queue as a result of issuing iprcv_dtq.	E_OK
An available space was secured in the data queue area of the target data queue as a result of issuing trcv_dtq.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 Data is written to the data queue area of the target data queue in the order of the data transmission request.

Note 2 Invoking tasks are queued to the transmission wait queue of the target data queue in the order defined during configuration (FIFO order or priority order).

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to snd_dtq will be executed. When TMO_POL is specified, processing equivalent to psnd_dtq /ipsnd_dtq will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tmout* < TMO_FEVR
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0x0 - *dtqid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified data queue is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Timeout. - Polling failure or timeout.

fsnd_dtq
ifsnd_dtq

Outline

Forced send to data queue.

C format

```
ER      fsnd_dtq (ID dtqid, VP_INT data);
ER      ifsnd_dtq (ID dtqid, VP_INT data);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue to which the data element is sent.
I	`VP_INT data;`	Data element to be sent to the data queue.

Explanation

These service calls write data specified by parameter *data* to the data queue area of the data queue specified by parameter *dtqid*.

If there is no available space for writing data in the data queue area of the target data queue when either of these service calls is issued, the service call overwrites data to the area with the oldest data that was written.

If a task has been queued to the reception wait queue of the target data queue when this service call is issued, this service call does not write data but transfers the data to the task. As a result, the task is unlinked from the reception wait queue and moves from the WAITING state (data reception wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0x0 - *dtqid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_ILUSE	-28	Illegal service call use. - The capacity of the data queue area is 0.
E_NOEXS	-42	Non-existent object. - Specified data queue is not registered.

rcv_dtq

Outline

Receive from data queue (waiting forever).

C format

```
ER      rcv_dtq (ID dtqid, VP_INT *p_data);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue from which a data element is received.
O	`VP_INT *p_data;`	Data element received from the data queue.

Explanation

This service call reads data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.

If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when this service call is issued, the service call does not read data but queues the invoking task to the reception wait queue of the target data queue and moves it from the RUNNING state to the WAITING state (data reception wait state).

The receiving WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

Receiving WAITING State for a Data Queue Cancel Operation	Return Value
Data was written to the data queue area of the target data queue as a result of issuing snd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing psnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing ipsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing tsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing fsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing ifsnd_dtq.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

Note 1 Invoking tasks are queued to the reception wait queue of the target data queue in the order of the data reception request.

Note 2 If the receiving

Note 3 for a data queue is forcibly released by issuing rel_wai or irel_wai, the contents of the area specified by parameter *p_data* will be undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0x0 - *dtqid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified data queue is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

prcv_dtq
iprcv_dtq

Outline

Receive from data queue (polling).

C format

```
ER      prcv_dtq (ID dtqid, VP_INT *p_data);
ER      iprcv_dtq (ID dtqid, VP_INT *p_data);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue from which a data element is received.
O	`VP_INT *p_data;`	Data element received from the data queue.

Explanation(s)

These service calls read data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.

If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when either of these service calls is issued, the service call does not read data but E_TMOUT is returned.

Note If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when either of these service calls is issued, the contents in the area specified by parameter *p_data* become undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0x0 - *dtqid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified data queue is not registered.
E_TMOUT	-50	Polling failure. - No data exists in the target data queue.

trcv_dtq

Outline

Receive from data queue (with timeout).

C format

```
ER      trcv_dtq (ID dtqid, VP_INT *p_data, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue from which a data element is received.
O	`VP_INT *p_data;`	Data element received from the data queue.
I	`TMO tmout;`	Specified timeout (unit:millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified timeout.

Explanation

This service call reads data in the data queue area of the data queue specified by parameter *dtqid* and stores it to the area specified by parameter *p_data*.

If no data could be read from the data queue area of the target data queue (no data has been written to the data queue area) when this service call is issued, the service call does not read data but queues the invoking task to the reception wait queue of the target data queue and moves it from the RUNNING state to the WAITING state with time out (data reception wait state).

The receiving WAITING state for a data queue is cancelled in the following cases, and then moved to the READY state.

Receiving WAITING State for a Data Queue Cancel Operation	Return Value
Data was written to the data queue area of the target data queue as a result of issuing snd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing psnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing ipsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing tsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing fsnd_dtq.	E_OK
Data was written to the data queue area of the target data queue as a result of issuing ifsnd_dtq.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

Note 1 Invoking tasks are queued to the reception wait queue of the target data queue in the order of the data reception request.

Note 2 If the data reception wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_data* become undefined.

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to rcv_dtq will be executed. When TMO_POL is specified, processing equivalent to prcv_dtq /iprcv_dtq will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tmout* < TMO_FEVR
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0x0 - *dtqid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified data queue is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Timeout. - Polling failure or timeout.

ref_dtq
iref_dtq

Outline

Reference data queue state.

C format

```
ER      ref_dtq (ID dtqid, T_RDTQ *pk_rdtq);
ER      iref_dtq (ID dtqid, T_RDTQ *pk_rdtq);
```

Parameter(s)

I/O	Parameter	Description
I	`ID dtqid;`	ID number of the data queue to be referenced.
O	`T_RDTQ *pk_rdtq;`	Pointer to the packet returning the data queue state.

[Data queue state packet: T_RDTQ]

```
typedef struct  t_rdtq {
    ID      stskid;         /*Existence of tasks waiting for data transmission*/
    ID      rtskid;         /*Existence of tasks waiting for data reception*/
    UINT    sdtqcnt;        /*Number of data elements in data queue*/
    ATR     dtqatr;         /*Attribute*/
    UINT    dtqcnt;         /*Data count*/
    ID      memid;          /*Reserved for future use*/
} T_RDTQ;
```

Explanation

These service calls store the detailed information of the data queue (existence of waiting tasks, number of data elements in the data queue, etc.) specified by parameter *dtqid* into the area specified by parameter *pk_rdtq*.

Note For details about the data queue state packet, refer to "15.2.5 Data queue state packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *dtqid* ≤ 0x0 - *dtqid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified data queue is not registered.

16.2.6 Synchronization and communication functions (mailboxes)

The following shows the service calls provided by the RI850V4 as the synchronization and communication functions (mailboxes).

Table 16-6 Synchronization and Communication Functions (Mailboxes)

Service Call	Function	Origin of Service Call
snd_mbx	Send to mailbox	Task, Non-task, Initialization routine
isnd_mbx	Send to mailbox	Task, Non-task, Initialization routine
rcv_mbx	Receive from mailbox (waiting forever)	Task
prcv_mbx	Receive from mailbox (polling)	Task, Non-task, Initialization routine
iprcv_mbx	Receive from mailbox (polling)	Task, Non-task, Initialization routine
trcv_mbx	Receive from mailbox (with timeout)	Task
ref_mbx	Reference mailbox state	Task, Non-task, Initialization routine
iref_mbx	Reference mailbox state	Task, Non-task, Initialization routine

snd_mbx
isnd_mbx

Outline

Send to mailbox.

C format

```
ER      snd_mbx (ID mbxid, T_MSG *pk_msg);
ER      isnd_mbx (ID mbxid, T_MSG *pk_msg);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbxid;`	ID number of the mailbox to which the message is sent.
I	`T_MSG *pk_msg;`	Start address of the message packet to be sent to the mailbox.

[Message packet: T_MSG]

```
typedef struct  t_msg {
    struct  t_msg   *msgnext;   /*Reserved for future use*/
} T_MSG;
```

[Message packet: T_MSG_PRI]

```
typedef struct  t_msg_pri {
    struct  t_msg   msgque;     /*Reserved for future use*/
    PRI     msgpri;             /*Message priority*/
} T_MSG_PRI;
```

Explanation

This service call transmits the message specified by parameter *pk_msg* to the mailbox specified by parameter *mbxid* (queues the message in the wait queue).

If a task is queued to the target mailbox wait queue when this service call is issued, the message is not queued but handed over to the relevant task (first task of the wait queue).

As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (receiving WAITING state for a mailbox) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 Messages are queued to the target mailbox wait queue in the order defined by queuing method during configuration (FIFO order or priority order).

Note 2 With the RI850V4 mailbox, only the start address of the message is handed over to the receiving processing program, but the message contents are not copied to a separate area. The message contents can therefore be rewritten even after this service call is issued.

Note 3 For details about the message packet, refer to "15.2.6 Message packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - msgpri ≤ 0x0 - msgpri > Maximum message priority
E_ID	-18	Invalid ID number. - *mbxid* ≤ 0x0 - *mbxid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified mailbox is not registered.

rcv_mbx

Outline

Receive from mailbox (waiting forever).

C format

```
ER      rcv_mbx (ID mbxid, T_MSG **ppk_msg);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbxid;`	ID number of the mailbox from which a message is received.
O	`T_MSG **ppk_msg;`	Start address of the message packet received from the mailbox.

[Message packet: T_MSG]

```
typedef struct  t_msg {
    struct  t_msg   *msgnext;   /*Reserved for future use*/
} T_MSG;
```

[Message packet: T_MSG_PRI]

```
typedef struct  t_msg_pri {
    struct  t_msg   msgque;     /*Reserved for future use*/
    PRI     msgpri;             /*Message priority*/
} T_MSG_PRI;
```

Explanation

This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.

If no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, this service call does not receive messages but queues the invoking task to the target mailbox wait queue and moves it from the RUNNING state to the WAITING state (message reception wait state).

The receiving WAITING state for a mailbox is cancelled in the following cases, and then moved to the READY state.

Receiving WAITING State for a Mailbox Cancel Operation	Return Value
A message was transmitted to the target mailbox as a result of issuing snd_mbx.	E_OK
A message was transmitted to the target mailbox as a result of issuing isnd_mbx.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

Note 1 Invoking tasks are queued to the target mailbox wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 If the receiving WAITING state for a mailbox is forcibly released by issuing rel_wai or irel_wai, the contents of the area specified by parameter *ppk_msg* will be undefined.

Note 3 For details about the message packet, refer to "15.2.6 Message packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mbxid* ≤ 0x0 - *mbxid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified mailbox is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

prcv_mbx
iprcv_mbx

Outline

Receive from mailbox (polling).

C format

```
ER      prcv_mbx (ID mbxid, T_MSG **ppk_msg);
ER      iprcv_mbx (ID mbxid, T_MSG **ppk_msg);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mbxid*;	ID number of the mailbox from which a message is received.
O	T_MSG **ppk_msg;	Start address of the message packet received from the mailbox.

[Message packet: T_MSG]

```
typedef struct  t_msg {
    struct  t_msg   *msgnext;   /*Reserved for future use*/
} T_MSG;
```

[Message packet: T_MSG_PRI]

```
typedef struct  t_msg_pri {
    struct  t_msg   msgque;     /*Reserved for future use*/
    PRI     msgpri;             /*Message priority*/
} T_MSG_PRI;
```

Explanation

This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.

If the message could not be received from the target mailbox (no messages were queued in the wait queue) when this service call is issued, message reception processing is not executed but "E_TMOUT" is returned.

Note 1 If no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, the contents in the area specified by parameter *ppk_msg* become undefined.

Note 2 For details about the message packet, refer to "15.2.6 Message packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.

Macro	Value	Description
E_ID	-18	Invalid ID number. - *mbxid* ≤ 0x0 - *mbxid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified mailbox is not registered.
E_TMOUT	-50	Polling failure. - No message exists in the target mailbox.

trcv_mbx

Outline

Receive from mailbox (with timeout).

C format

```
ER      trcv_mbx (ID mbxid, T_MSG **ppk_msg, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbxid;`	ID number of the mailbox from which a message is received.
O	`T_MSG **ppk_msg;`	Start address of the message packet received from the mailbox.
I	`TMO tmout;`	Specified timeout (unit:millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified timeout.

[Message packet: T_MSG]

```
typedef struct  t_msg {
    struct  t_msg   *msgnext;   /*Reserved for future use*/
} T_MSG;
```

[Message packet: T_MSG_PRI]

```
typedef struct  t_msg_pri {
    struct  t_msg   msgque;     /*Reserved for future use*/
    PRI     msgpri;             /*Message priority*/
} T_MSG_PRI;
```

Explanation

This service call receives a message from the mailbox specified by parameter *mbxid*, and stores its start address in the area specified by parameter *ppk_msg*.

If no message could be received from the target mailbox (no messages were queued to the wait queue) when this service call is issued, this service call does not receive messages but queues the invoking task to the target mailbox wait queue and moves it from the RUNNING state to the WAITING state with timeout (message reception wait state).

The receiving WAITING state for a mailbox is cancelled in the following cases, and then moved to the READY state.

Receiving WAITING State for a Mailbox Cancel Operation	Return Value
A message was transmitted to the target mailbox as a result of issuing snd_mbx.	E_OK
A message was transmitted to the target mailbox as a result of issuing isnd_mbx.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI

Receiving WAITING State for a Mailbox Cancel Operation	Return Value
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

Note 1 Invoking tasks are queued to the target mailbox wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 If the message reception wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *ppk_msg* become undefined.

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to rcv_mbx will be executed. When TMO_POL is specified, processing equivalent to prcv_mbx /iprcv_mbx will be executed.

Note 4 For details about the message packet, refer to "15.2.6 Message packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tmout* < TMO_FEVR
E_ID	-18	Invalid ID number. - *mbxid* ≤ 0x0 - *mbxid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified mailbox is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Timeout. - Polling failure or timeout.

ref_mbx
iref_mbx

Outline

Reference mailbox state.

C format

```
ER      ref_mbx (ID mbxid, T_RMBX *pk_rmbx);
ER      iref_mbx (ID mbxid, T_RMBX *pk_rmbx);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mbxid;`	ID number of the mailbox to be referenced.
O	`T_RMBX *pk_rmbx;`	Pointer to the packet returning the mailbox state.

[Mailbox state packet: T_RMBX]

```
typedef struct  t_rmbx {
    ID      wtskid;         /*Existence of waiting task*/
    T_MSG   *pk_msg;        /*Existence of waiting message*/
    ATR     mbxatr;         /*Attribute*/
} T_RMBX;
```

Explanation

Stores mailbox state packet (ID number of the task at the head of the wait queue, start address of the message packet at the head of the wait queue) of the mailbox specified by parameter *mbxid* in the area specified by parameter *pk_rmbx*.

Note For details about the mailbox state packet, refer to "15.2.7 Mailbox state packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mbxid* ≤ 0x0 - *mbxid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified mailbox is not registered.

16.2.7 Extended synchronization and communication functions (mutexes)

The following shows the service calls provided by the RI850V4 as the extended synchronization and communication functions (mutexes).

Table 16-7 Extended Synchronization and Communication Functions (Mutexes)

Service Call	Function	Origin of Service Call
loc_mtx	Lock mutex (waiting forever)	Task
ploc_mtx	Lock mutex (polling)	Task
tloc_mtx	Lock mutex (with timeout)	Task
unl_mtx	Unlock mutex	Task
ref_mtx	Reference mutex state	Task, Non-task, Initialization routine
iref_mtx	Reference mutex state	Task, Non-task, Initialization routine

loc_mtx

Outline

Lock mutex (waiting forever).

C format

```
ER      loc_mtx (ID mtxid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mtxid*;	ID number of the mutex to be locked.

Explanation

This service call locks the mutex specified by parameter *mtxid*.

If the target mutex could not be locked (another task has been locked) when this service call is issued, this service call queues the invoking task to the target mutex wait queue and moves it from the RUNNING state to the WAITING state (mutex wait state).

The WAITING state for a mutex is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Mutex Cancel Operation	Return Value
The locked state of the target mutex was cancelled as a result of issuing unl_mtx.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ext_tsk.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ter_tsk.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

Note 1 Invoking tasks are queued to the target mutex wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 In the RI850V4, E_ILUSE is returned if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mtxid* ≤ 0x0 - *mtxid* > Maximum ID number

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_ILUSE	-28	Illegal service call use. - Multiple locking of a mutex.
E_NOEXS	-42	Non-existent object. - Specified mutex is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

ploc_mtx

Outline

Lock mutex (polling).

C format

```
ER      ploc_mtx (ID mtxid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mtxid;`	ID number of the mutex to be locked.

Explanation

This service call locks the mutex specified by parameter *mtxid*.
If the target mutex could not be locked (another task has been locked) when this service call is issued but E_TMOUT is returned.

Note In the RI850V4, E_ILUSE is returned if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mtxid* ≤ 0x0 - *mtxid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state.
E_ILUSE	-28	Illegal service call use. - Multiple locking of a mutex.
E_NOEXS	-42	Non-existent object. - Specified mutex is not registered.
E_TMOUT	-50	Polling failure. - The target mutex has been locked by another task.

tloc_mtx

Outline

Lock mutex (with timeout).

C format

```
ER      tloc_mtx (ID mtxid, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mtxid*;	ID number of the mutex to be locked.
I	TMO *tmout*;	Specified timeout (unit:millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified timeout.

Explanation

This service call locks the mutex specified by parameter *mtxid*.

If the target mutex could not be locked (another task has been locked) when this service call is issued, this service call queues the invoking task to the target mutex wait queue and moves it from the RUNNING state to the WAITING state with timeout (mutex wait state).

The WAITING state for a mutex is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Mutex Cancel Operation	Return Value
The locked state of the target mutex was cancelled as a result of issuing unl_mtx.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ext_tsk.	E_OK
The locked state of the target mutex was cancelled as a result of issuing ter_tsk.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

Note 1 Invoking tasks are queued to the target mutex wait queue in the order defined during configuration (FIFO order or priority order).

Note 2 In the RI850V4, E_ILUSE is returned if this service call is re-issued for the mutex that has been locked by the invoking task (multiple-locking of mutex).

Note 3 TMO_FEVR is specified for wait time *tmout*, processing equivalent to loc_mtx will be executed. When TMO_POL is specified, processing equivalent to ploc_mtx will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tmout* < TMO_FEVR
E_ID	-18	Invalid ID number. - *mtxid* ≤ 0x0 - *mtxid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_ILUSE	-28	Illegal service call use. - Multiple locking of a mutex.
E_NOEXS	-42	Non-existent object. - Specified mutex is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Timeout. - Polling failure or timeout.

unl_mtx

Outline

Unlock mutex.

C format

```
ER      unl_mtx (ID mtxid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mtxid*;	ID number of the mutex to be unlocked.

Explanation

This service call unlocks the locked mutex specified by parameter *mtxid*.

If a task has been queued to the target mutex wait queue when this service call is issued, mutex lock processing is performed by the task (the first task in the wait queue) immediately after mutex unlock processing.

As a result, the task is unlinked from the wait queue and moves from the WAITING state (mutex wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note A locked mutex can be unlocked only by the task that locked the mutex.
If this service call is issued for a mutex that was not locked by an invoking task, no processing is performed but E_ILUSE is returned.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mtxid* ≤ 0x0 - *mtxid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state.
E_ILUSE	-28	Illegal service call use. - Multiple unlocking of a mutex. - The invoking task does not have the specified mutex locked.
E_NOEXS	-42	Non-existent object. - Specified mutex is not registered.

ref_mtx
iref_mtx

Outline

Reference mutex state.

C format

```
ER      ref_mtx (ID mtxid, T_RMTX *pk_rmtx);
ER      iref_mtx (ID mtxid, T_RMTX *pk_rmtx);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mtxid;`	ID number of the mutex to be referenced.
O	`T_RMTX *pk_rmtx;`	Pointer to the packet returning the mutex state.

[Mutex state packet: T_RMTX]

```
typedef struct  t_rmtx {
    ID      htskid;         /*Existence of locked mutex*/
    ID      wtskid;         /*Existence of waiting task*/
    ATR     mtxatr;         /*Attribute*/
    PRI     ceilpri;        /*Reserved for future use*/
} T_RMTX;
```

Explanation

The service calls store the detailed information of the mutex specified by parameter *mtxid* (existence of locked mutexes, waiting tasks, etc.) into the area specified by parameter *pk_rmtx*.

Note For details about the mutex state packet, refer to "15.2.8 Mutex state packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mtxid* ≤ 0x0 - *mtxid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified mutex is not registered.

16.2.8 Memory pool management functions (fixed-sized memory pools)

The following shows the service calls provided by the RI850V4 as the memory pool management functions (fixed-sized memory pools).

Table 16-8 Memory Pool Management Functions (Fixed-Sized Memory Pools)

Service Call	Function	Origin of Service Call
get_mpf	Acquire fixed-sized memory block (waiting forever)	Task
pget_mpf	Acquire fixed-sized memory block (polling)	Task, Non-task, Initialization routine
ipget_mpf	Acquire fixed-sized memory block (polling)	Task, Non-task, Initialization routine
tget_mpf	Acquire fixed-sized memory block (with timeout)	Task
rel_mpf	Release fixed-sized memory block	Task, Non-task, Initialization routine
irel_mpf	Release fixed-sized memory block	Task, Non-task, Initialization routine
ref_mpf	Reference fixed-sized memory pool state	Task, Non-task, Initialization routine
iref_mpf	Reference fixed-sized memory pool state	Task, Non-task, Initialization routine

get_mpf

Outline

Acquire fixed-sized memory block (waiting forever).

C format

```
ER      get_mpf (ID mpfid, VP *p_blk);
```

Parameter(s)

I/O	Parameter	Description
I	ID mpfid;	ID number of the fixed-sized memory pool from which a memory block is acquired.
O	VP *p_blk;	Start address of the acquired memory block.

Explanation

This service call acquires the fixed-sized memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.

If no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, this service call does not acquire the fixed-size memory block but queues the invoking task to the target fixed-size memory pool wait queue and moves it from the RUNNING state to the WAITING state (fixed-size memory block acquisition wait state).

The WAITING state for a fixed-sized memory block is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Fixed-sized Memory Block Cancel Operation	Return Value
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf.	E_OK
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing irel_mpf.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

Note 1 The RI850V4 does not perform memory clear processing when acquiring a fixed-sized memory block. The contents of the acquired fixed-sized memory block are therefore undefined.

Note 2 Invoking tasks are queued to the target fixed-size memory pool wait queue in the order defined during configuration (FIFO order or priority order).

Note 3 If the fixed-size memory block acquisition wait state is cancelled because rel_wai or irel_wai was issued, the contents in the area specified by parameter *p_blk* become undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0x0 - *mpfid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified fixed-sized memory pool is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

pget_mpf
ipget_mpf

Outline

Acquire fixed-sized memory block (polling).

C format

```
ER      pget_mpf (ID mpfid, VP *p_blk);
ER      ipget_mpf (ID mpfid, VP *p_blk);
```

Parameter(s)

I/O	Parameter		Description
I	ID	*mpfid*;	ID number of the fixed-sized memory pool from which a memory block is acquired.
O	VP	**p_blk*;	Start address of the acquired memory block.

Explanation

This service call acquires the fixed-sized memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.

If a fixed-sized memory block could not be acquired from the target fixed-sized memory pool (no available fixed-sized memory blocks exist) when this service call is issued, fixed-sized memory block acquisition processing is not performed but "E_TMOUT" is returned.

Note 1 The RI850V4 does not perform memory clear processing when acquiring a fixed-sized memory block. The contents of the acquired fixed-sized memory block are therefore undefined.

Note 2 If no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, the contents in the area specified by parameter *p_blk* become undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0x0 - *mpfid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified fixed-sized memory pool is not registered.

Macro	Value	Description
E_TMOUT	-50	Polling failure. - There is no free memory block in the target fixed-sized memory pool.

tget_mpf

Outline

Acquire fixed-sized memory block (with timeout).

C format

```
ER      tget_mpf (ID mpfid, VP *p_blk, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mpfid;`	ID number of the fixed-sized memory pool from which a memory block is acquired.
O	`VP *p_blk;`	Start address of the acquired memory block.
I	`TMO tmout;`	Specified timeout (unit:millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified timeout.

Explanation

This service call acquires the fixed-sized memory block from the fixed-sized memory pool specified by parameter *mpfid* and stores the start address in the area specified by parameter *p_blk*.

If no fixed-size memory blocks could be acquired from the target fixed-size memory pool (no available fixed-size memory blocks exist) when this service call is issued, this service call does not acquire the fixed-size memory block but queues the invoking task to the target fixed-size memory pool wait queue and moves it from the RUNNING state to the WAITING state with timeout (fixed-size memory block acquisition wait state).

The WAITING state for a fixed-sized memory block is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Fixed-sized Memory Block Cancel Operation	Return Value
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing rel_mpf.	E_OK
A fixed-sized memory block was returned to the target fixed-sized memory pool as a result of issuing irel_mpf.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

Note 1 The RI850V4 does not perform memory clear processing when acquiring a fixed-sized memory block. The contents of the acquired fixed-sized memory block are therefore undefined.

Note 2 If no fixed-sized memory blocks can be acquired from the target fixed-sized memory pool (no available fixed-sized memory blocks exist) when this service call is issued, the contents in the area specified by parameter *p_blk* become undefined.

Note 3 If the fixed-size memory block acquisition wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_blk* become undefined.

Note 4 TMO_FEVR is specified for wait time *tmout*, processing equivalent to get_mpf will be executed. When TMO_POL is specified, processing equivalent to pget_mpf /ipget_mpf will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tmout* < TMO_FEVR
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0x0 - *mpfid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified fixed-sized memory pool is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Timeout. - Polling failure or timeout.

rel_mpf
irel_mpf

Outline

Release fixed-sized memory block.

C format

```
ER      rel_mpf (ID mpfid, VP blk);
ER      irel_mpf (ID mpfid, VP blk);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mpfid*;	ID number of the fixed-sized memory pool to which the memory block is released.
I	VP *blk*;	Start address of the memory block to be released.

Explanation

This service call returns the fixed-sized memory block specified by parameter *blk* to the fixed-sized memory pool specified by parameter *mpfid*.

If a task is queued to the target fixed-sized memory pool wait queue when this service call is issued, fixed-sized memory block return processing is not performed but fixed-sized memory blocks are returned to the relevant task (first task of wait queue).

As a result, the relevant task is unlinked from the wait queue and is moved from the WAITING state (WAITING state for a fixed-sized memory block) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 The RI850V4 does not perform memory clear processing when returning the acquired fixed-size memory block. The contents of the returned fixed-size memory block are therefore undefined.

Note 2 When returning fixed-size memory blocks, be sure to issue either of these service calls for the acquired fixed-size memory pools. If the service call is issued for another fixed-size memory pool, no error results but the operation is not guaranteed after that.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0x0 - *mpfid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified fixed-sized memory pool is not registered.

ref_mpf
iref_mpf

Outline

Reference fixed-sized memory pool state.

C format

```
ER      ref_mpf (ID mpfid, T_RMPF *pk_rmpf);
ER      iref_mpf (ID mpfid, T_RMPF *pk_rmpf);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mpfid;`	ID number of the fixed-sized memory pool to be referenced.
O	`T_RMPF *pk_rmpf;`	Pointer to the packet returning the fixed-sized memory pool state.

[Fixed-sized memory pool state packet: T_RMPF]

```
typedef struct  t_rmpf {
    ID      wtskid;         /*Existence of waiting task*/
    UINT    fblkcnt;        /*Number of free memory blocks*/
    ATR     mpfatr;         /*Attribute*/
    ID      memid;          /*Reserved for future use*/
} T_RMPF;
```

Explanation

Stores fixed-sized memory pool state packet (ID number of the task at the head of the wait queue, number of free memory blocks, etc.) of the fixed-sized memory pool specified by parameter *mpfid* in the area specified by parameter *pk_rmpf*.

Note For details about the fixed-sized memory pool state packet, refer to "15.2.9 Fixed-sized memory pool state packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mpfid* ≤ 0x0 - *mpfid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified fixed-sized memory pool is not registered.

16.2.9 Memory pool management functions (variable-sized memory pools)

The following shows the service calls provided by the RI850V4 as the memory pool management functions (variable-sized memory pools).

Table 16-9 Memory Pool Management Functions (Variable-Sized Memory Pools)

Service Call	Function	Origin of Service Call
get_mpl	Acquire variable-sized memory block (waiting forever)	Task
pget_mpl	Acquire variable-sized memory block (polling)	Task, Non-task, Initialization routine
ipget_mpl	Acquire variable-sized memory block (polling)	Task, Non-task, Initialization routine
tget_mpl	Acquire variable-sized memory block (with timeout)	Task
rel_mpl	Release variable-sized memory block	Task, Non-task, Initialization routine
irel_mpl	Release variable-sized memory block	Task, Non-task, Initialization routine
ref_mpl	Reference variable-sized memory pool state	Task, Non-task, Initialization routine
iref_mpl	Reference variable-sized memory pool state	Task, Non-task, Initialization routine

get_mpl

Outline

Acquire variable-sized memory block (waiting forever).

C format

```
ER      get_mpl (ID mplid, UINT blksz, VP *p_blk);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mplid;`	ID number of the variable-sized memory pool from which a memory block is acquired.
I	`UINT blksz;`	Memory block size to be acquired (in bytes).
O	`VP *p_blk;`	Start address of the acquired memory block.

Explanation

This service call acquires a variable-size memory block of the size specified by parameter blksz from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.

If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory blocks but queues the invoking task to the target variable-size memory pool wait queue and moves it from the RUNNING state to the WAITING state (variable-size memory block acquisition wait state).

The WAITING state for a variable-sized memory block is cancelled in the following cases, and then moved to the READY state

WAITING State for a Variable-sized Memory Block Cancel Operation	Return Value
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing rel_mpl.	E_OK
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing irel_mpl.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI

Note 1 The RI850V4 acquires variable-size memory blocks in the unit of "integral multiple of 4". If a value other than an integral multiple of 4 is specified for parameter *blksz*, it is rounded up to be an integral multiple of 4.

Note 2 Invoking tasks are queued to the target variable-size memory pool wait queue in the order defined during configuration (FIFO order or priority order).

Note 3 If the variable-size memory block acquisition wait state is cancelled because rel_wai or irel_wai was issued, the contents in the area specified by parameter *p_blk* become undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *blksz* = 0x0 - *blksz* > 0x7fffffff
E_ID	-18	Invalid ID number. - *mplid* ≤ 0x0 - *mplid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified variable-sized memory pool is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.

pget_mpl
ipget_mpl

Outline

Acquire variable-sized memory block (polling).

C format

```
ER      pget_mpl (ID mplid, UINT blksz, VP *p_blk);
ER      ipget_mpl (ID mplid, UINT blksz, VP *p_blk);
```

Parameter(s)

I/O	Parameter		Description
I	ID	*mplid*;	ID number of the variable-sized memory pool from which a memory block is acquired.
I	UINT	*blksz*;	Memory block size to be acquired (in bytes).
O	VP	**p_blk*;	Start address of the acquired memory block.

Explanation

This service call acquires a variable-size memory block of the size specified by parameter *blksz* from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.

If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory block but returns E_TMOUT.

Note 1 The RI850V4 acquires variable-size memory blocks in the unit of "integral multiple of 4". If a value other than an integral multiple of 4 is specified for parameter *blksz*, it is rounded up to be an integral multiple of 4.

Note 2 If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, the contents in the area specified by parameter *p_blk* become undefined.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *blksz* = 0x0 - *blksz* > 0x7fffffff
E_ID	-18	Invalid ID number. - *mplid* ≤ 0x0 - *mplid* > Maximum ID number

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified variable-sized memory pool is not registered.
E_TMOUT	-50	Polling failure. - No successive areas equivalent to the requested size were available in the target variable-size memory pool.

tget_mpl

Outline

Acquire variable-sized memory block (with timeout).

C format

```
ER      tget_mpl (ID mplid, UINT blksz, VP *p_blk, TMO tmout);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mplid;`	ID number of the variable-sized memory pool from which a memory block is acquired.
I	`UINT blksz;`	Memory block size to be acquired (in bytes).
O	`VP *p_blk;`	Start address of the acquired memory block.
I	`TMO tmout;`	Specified timeout (unit:millisecond). TMO_FEVR: Waiting forever. TMO_POL: Polling. Value: Specified timeout.

Explanation

This service call acquires a variable-size memory block of the size specified by parameter *blksz* from the variable-size memory pool specified by parameter *mplid*, and stores its start address into the area specified by parameter *p_blk*.

If no variable-size memory blocks could be acquired from the target variable-size memory pool (no successive areas equivalent to the requested size were available) when this service call is issued, this service call does not acquire variable-size memory blocks but queues the invoking task to the target variable-size memory pool wait queue and moves it from the RUNNING state to the WAITING state with timeout (variable-size memory block acquisition wait state).

The WAITING state for a variable-sized memory block is cancelled in the following cases, and then moved to the READY state.

WAITING State for a Variable-sized Memory Block Cancel Operation	Return Value
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing rel_mpl.	E_OK
The variable-size memory block that satisfies the requested size was returned to the target variable-size memory pool as a result of issuing irel_mpl.	E_OK
Forced release from waiting (accept rel_wai while waiting).	E_RLWAI
Forced release from waiting (accept irel_wai while waiting).	E_RLWAI
Polling failure or timeout.	E_TMOUT

Note 1 The RI850V4 acquires variable-size memory blocks in the unit of "integral multiple of 4". If a value other than an integral multiple of 4 is specified for parameter *blksz*, it is rounded up to be an integral multiple of 4.

Note 2 Invoking tasks are queued to the target variable-size memory pool wait queue in the order defined during configuration (FIFO order or priority order).

Note 3 If the variable-size memory block acquisition wait state is cancelled because rel_wai or irel_wai was issued or the wait time elapsed, the contents in the area specified by parameter *p_blk* become undefined.

Note 4 TMO_FEVR is specified for wait time *tmout*, processing equivalent to get_mpl will be executed. When TMO_POL is specified, processing equivalent to pget_mpl /ipget_mpl will be executed.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *blksz* = 0x0 - *blksz* > 0x7fffffff - *tmout* < TMO_FEVR
E_ID	-18	Invalid ID number. - *mplid* ≤ 0x0 - *mplid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.
E_NOEXS	-42	Non-existent object. - Specified variable-sized memory pool is not registered.
E_RLWAI	-49	Forced release from the WAITING state. - Accept rel_wai/irel_wai while waiting.
E_TMOUT	-50	Timeout. - Polling failure or timeout.

rel_mpl
irel_mpl

Outline

Release variable-sized memory block.

C format

```
ER      rel_mpl (ID mplid, VP blk);
ER      irel_mpl (ID mplid, VP blk);
```

Parameter(s)

I/O	Parameter	Description
I	ID *mplid*;	ID number of the variable-sized memory pool to which the memory block is released.
I	VP *blk*;	Start address of memory block to be released.

Explanation

This service call returns the variable-sized memory block specified by parameter *blk* to the variable-sized memory pool specified by parameter *mplid*.

After returning the variable-size memory blocks, these service calls check the tasks queued to the target variable-size memory pool wait queue from the top, and assigns the memory if the size of memory requested by the wait queue is available. This operation continues until no tasks queued to the wait queue remain or no memory space is available. As a result, the task that acquired the memory is unlinked from the queue and moved from the WAITING state (variable-size memory block acquisition wait state) to the READY state, or from the WAITING-SUSPENDED state to the SUSPENDED state.

Note 1 The RI850V4 does not perform memory clear processing when returning the acquired variable-size memory block. The contents of the returned variable-size memory block are therefore undefined.

Note 2 When returning variable-size memory blocks, be sure to issue either of these service calls for the acquired variable-size memory pools. If the service call is issued for another variable-size memory pool, no error results but the operation is not guaranteed after that.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mplid* ≤ 0x0 - *mplid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified variable-sized memory pool is not registered.

ref_mpl
iref_mpl

Outline

Reference variable-sized memory pool state.

C format

```
ER      ref_mpl (ID mplid, T_RMPL *pk_rmpl);
ER      iref_mpl (ID mplid, T_RMPL *pk_rmpl);
```

Parameter(s)

I/O	Parameter	Description
I	`ID mplid;`	ID number of the variable-sized memory pool to be referenced.
O	`T_RMPL *pk_rmpl;`	Pointer to the packet returning the variable-sized memory pool state.

[Variable-sized memory pool state packet: T_RMPL]

```
typedef struct  t_rmpl {
    ID      wtskid;         /*Existence of waiting task*/
    SIZE    fmplsz;         /*Total size of free memory blocks*/
    UINT    fblksz;         /*Maximum memory block size available*/
    ATR     mplatr;         /*Attribute*/
    ID      memid;          /*Reserved for future use*/
} T_RMPL;
```

Explanation

These service calls store the detailed information (ID number of the task at the head of the wait queue, total size of free memory blocks, etc.) of the variable-size memory pool specified by parameter *mplid* into the area specified by parameter *pk_rmpl*.

Note For details about the variable-sized memory pool state packet, refer to "15.2.10 Variable-sized memory pool state packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *mplid* ≤ 0x0 - *mplid* > Maximum ID number

Macro	Value	Description
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified variable-sized memory pool is not registered.

16.2.10 Time management functions

The following shows the service calls provided by the RI850V4 as the time management functions.

Table 16-10 Time Management Functions

Service Call	Function	Origin of Service Call
set_tim	Set system time	Task, Non-task, Initialization routine
iset_tim	Set system time	Task, Non-task, Initialization routine
get_tim	Reference system time	Task, Non-task, Initialization routine
iget_tim	Reference system time	Task, Non-task, Initialization routine
sta_cyc	Start cyclic handler operation	Task, Non-task, Initialization routine
ista_cyc	Start cyclic handler operation	Task, Non-task, Initialization routine
stp_cyc	Stop cyclic handler operation	Task, Non-task, Initialization routine
istp_cyc	Stop cyclic handler operation	Task, Non-task, Initialization routine
ref_cyc	Reference cyclic handler state	Task, Non-task, Initialization routine
iref_cyc	Reference cyclic handler state	Task, Non-task, Initialization routine

set_tim
iset_tim

Outline

Set system time.

C format

```
ER      set_tim (SYSTIM *p_systim);
ER      iset_tim (SYSTIM *p_systim);
```

Parameter(s)

I/O	Parameter	Description
I	SYSTIM *p_systim;	Time to set as system time.

[System time packet: SYSTIM]

```
typedef struct  t_systim {
    UW      ltime;          /*System time (lower 32 bits)*/
    UH      utime;          /*System time (higher 16 bits)*/
} SYSTIM;
```

Explanation

These service calls change the RI850V4 system time (unit: millisecond) to the time specified by parameter *p_systim*.

Note For details about the system time packet, refer to "15.2.11 System time packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

get_tim
iget_tim

Outline

Reference system time.

C format

```
ER      get_tim (SYSTIM *p_systim);
ER      iget_tim (SYSTIM *p_systim);
```

Parameter(s)

I/O	Parameter	Description
O	SYSTIM *p_systim;	Current system time.

[System time packet: SYSTIM]

```
typedef struct  t_systim {
    UW      ltime;          /*System time (lower 32 bits)*/
    UH      utime;          /*System time (higher 16 bits)*/
} SYSTIM;
```

Explanation

These service calls store the RI850V4 system time (unit: millisecond) into the area specified by parameter *p_systim*.

Note 1 The RI850V4 ignores the numeric values that cannot be expressed as the system time (values overflowed from the 48-bit width).

Note 2 For details about the system time packet, refer to "15.2.11 System time packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

sta_cyc
ista_cyc

Outline

Start cyclic handler operation.

C format

```
ER      sta_cyc (ID cycid);
ER      ista_cyc (ID cycid);
```

Parameter(s)

I/O	Parameter	Description
I	ID *cycid*;	ID number of the cyclic handler operation to be started.

Explanation

This service call moves the cyclic handler specified by parameter *cycid* from the non-operational state (STP state) to operational state (STA state).

As a result, the target cyclic handler is handled as an activation target of the RI850V4.

The relative interval from when either of this service call is issued until the first activation request is issued varies depending on whether the TA_PHS attribute is specified for the target cyclic handler during configuration.

- If the TA_PHS attribute is specified
  The target cyclic handler activation timing is set based on the activation phases (initial activation phase *cycphs* and activation cycle *cyctim*) defined during configuration.
  If the target cyclic handler has already been started, however, no processing is performed even if this service call is issued, but it is not handled as an error.

- If the TA_PHS attribute is not specified
  The target cyclic handler activation timing is set based on the activation phase (activation cycle *cyctim*) when this service call is issued.
  This setting is performed regardless of the operating status of the target cyclic handler.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *cycid* ≤ 0x0 - *cycid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified cyclic handler is not registered.

stp_cyc
istp_cyc

Outline

Stop cyclic handler operation.

C format

```
ER      stp_cyc (ID cycid);
ER      istp_cyc (ID cycid);
```

Parameter(s)

I/O	Parameter	Description
I	`ID cycid;`	ID number of the cyclic handler operation to be stopped.

Explanation

This service call moves the cyclic handler specified by parameter *cycid* from the operational state (STA state) to non-operational state (STP state).

As a result, the target cyclic handler is excluded from activation targets of the RI850V4 until issue of sta_cyc or ista_cyc.

Note This service call does not perform queuing of stop requests. If the target cyclic handler has been moved to the non-operational state (STP state), therefore, no processing is performed but it is not handled as an error.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *cycid* ≤ 0x0 - *cycid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.
E_NOEXS	-42	Non-existent object. - Specified cyclic handler is not registered.

ref_cyc
iref_cyc

Outline

Reference cyclic handler state.

C format

```
ER      ref_cyc (ID cycid, T_RCYC *pk_rcyc);
ER      iref_cyc (ID cycid, T_RCYC *pk_rcyc);
```

Parameter(s)

I/O	Parameter	Description
I	`ID cycid;`	ID number of the cyclic handler to be referenced.
O	`T_RCYC *pk_rcyc;`	Pointer to the packet returning the cyclic handler state.

[Cyclic handler state packet: T_RCYC]

```
typedef struct  t_rcyc {
    STAT    cycstat;        /*Current state*/
    RELTIM  lefttim;        /*Time left before the next activation*/
    ATR     cycatr;         /*Attribute*/
    RELTIM  cyctim;         /*Activation cycle*/
    RELTIM  cycphs;         /*Activation phase*/
} T_RCYC;
```

Explanation

Stores cyclic handler state packet (current state, time left before the next activation, etc.) of the cyclic handler specified by parameter *cycid* in the area specified by parameter *pk_rcyc*.

Note For details about the cyclic handler state packet, refer to "15.2.12 Cyclic handler state packet".

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_ID	-18	Invalid ID number. - *cycid* ≤ 0x0 - *cycid* > Maximum ID number
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

Macro	Value	Description
E_NOEXS	-42	Non-existent object. - Specified cyclic handler is not registered.

16.2.11 System state management functions

The following shows the service calls provided by the RI850V4 as the system state management functions.

Table 16-11 System State Management Functions

Service Call	Function	Origin of Service Call
rot_rdq	Rotate task precedence	Task, Non-task, Initialization routine
irot_rdq	Rotate task precedence	Task, Non-task, Initialization routine
vsta_sch	Forced scheduler activation	Task
get_tid	Reference task ID in the RUNNING state	Task, Non-task, Initialization routine
iget_tid	Reference task ID in the RUNNING state	Task, Non-task, Initialization routine
loc_cpu	Lock the CPU	Task, Non-task
iloc_cpu	Lock the CPU	Task, Non-task
unl_cpu	Unlock the CPU	Task, Non-task
iunl_cpu	Unlock the CPU	Task, Non-task
sns_loc	Reference CPU state	Task, Non-task, Initialization routine
dis_dsp	Disable dispatching	Task
ena_dsp	Enable dispatching	Task
sns_dsp	Reference dispatching state	Task, Non-task, Initialization routine
sns_ctx	Reference contexts	Task, Non-task, Initialization routine
sns_dpn	Reference dispatching pending state	Task, Non-task, Initialization routine

rot_rdq
irot_rdq

Outline

Rotate task precedence.

C format

```
ER      rot_rdq (PRI tskpri);
ER      irot_rdq (PRI tskpri);
```

Parameter(s)

I/O	Parameter	Description
I	PRI tskpri;	Priority of the tasks whose precedence is rotated. TPRI_SELF: Current priority of the invoking task. Value: Priority of the tasks whose precedence is rotated.

Explanation

This service call re-queues the first task of the ready queue corresponding to the priority specified by parameter *tskpri* to the end of the queue to change the task execution order explicitly.

Note 1 This service call does not perform queuing of rotation requests. If no task is queued to the ready queue corresponding to the relevant priority, therefore, no processing is performed but it is not handled as an error.

Note 2 Round-robin scheduling can be implemented by issuing this service call via a cyclic handler in a constant cycle.

Note 3 The ready queue is a hash table that uses priority as the key, and tasks that have entered an executable state (READY state or RUNNING state) are queued in FIFO order.
Therefore, the scheduler realizes the RI850V4's scheduling system by executing task detection processing from the highest priority level of the ready queue upon activation, and upon detection of queued tasks, giving the CPU use right to the first task of the proper priority level.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_PAR	-17	Parameter error. - *tskpri* < 0x0 - *tskpri* > Maximum priority - When this service call was issued from a non-task, TPRI_SELF was specified *tskpri*.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

vsta_sch

Outline

Forced scheduler activation.

C format

```
ER      vsta_sch (void);
```

Parameter(s)

None.

Explanation

This service call explicitly forces the RI850V4 scheduler to activate. If a scheduling request has been kept pending, task switching may therefore occur.

Note The RI850V4 provides this service call as a function to activate a scheduler from a task for which preempt acknowledge status disable is defined during configuration.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state. - This service call was issued in the dispatching disabled state.

get_tid
iget_tid

Outline

Reference task ID in the RUNNING state.

C format

```
ER      get_tid (ID *p_tskid);
ER      iget_tid (ID *p_tskid);
```

Parameter(s)

I/O	Parameter	Description
O	`ID *p_tskid;`	ID number of the task in the RUNNING state.

Explanation

These service calls store the ID of a task in the RUNNING state in the area specified by parameter *p_tskid*.

Note This service call stores TSK_NONE in the area specified by parameter *p_tskid* if no tasks that have entered the RUNNING state exist (all tasks in the IDLE state).

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued in the CPU locked state.

loc_cpu
iloc_cpu

Outline

Lock the CPU.

C format

```
ER      loc_cpu (void);
ER      iloc_cpu (void);
```

Parameter(s)

None.

Explanation

These service calls change the system status type to the CPU locked state.
As a result, "EI level maskable interrupt acceptance" and "service call issue (except for some service calls)" are prohibited during the interval from when this service call is issued until unl_cpu or iunl_cpu is issued.

Service Call	Function
loc_cpu, iloc_cpu	Lock the CPU.
unl_cpu, iunl_cpu	Unlock the CPU.
sns_loc	Reference CPU state.
sns_dsp	Reference dispatching state.
sns_ctx	Reference contexts.
sns_dpn	Reference dispatch pending state.

If an EI level maskable interrupt is created during this period, the RI850V4 delays transition to the relevant interrupt processing (interrupt handler) until either unl_cpu or iunl_cpu is issued.

Note 1 The CPU locked state changed by issuing this service call must be cancelled before the processing program that issued this service call ends.

Note 2 This service call does not perform queuing of lock requests. If the system is in the CPU locked state, therefore, no processing is performed but it is not handled as an error.

Note 3 This service call manipulates PM*n* bits in the priority mask register (PMR) to disable acceptance of EI level maskable interrupts.
The PM*n* bits to be manipulated correspond to the interrupt priority range defined as the Maximum interrupt priority: maxintpri during configuration.
This service call does not manipulate the ID bit in the program status word (PSW).

Note 4 The RI850V4 realizes the TIME MANAGEMENT FUNCTIONS by using base clock timer interrupts that occur at constant intervals. If acknowledgment of the relevant base clock timer interrupt is disabled by issuing this service call, the TIME MANAGEMENT FUNCTIONS may no longer operate normally.

Note 5 If this service call or a service call other than sns_*xxx* is issued from when this service call is issued until unl_cpu or iunl_cpu is issued, the RI850V4 returns E_CTX.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

unl_cpu
iunl_cpu

Outline

Unlock the CPU.

C format

```
ER      unl_cpu (void);
ER      iunl_cpu (void);
```

Parameter(s)

None.

Explanation

These service calls change the system status from the CPU locked state to the CPU unlocked state.

As a result, "EI level maskable interrupt acceptance" and "service call issue" restricted (prohibited) through issue of loc_cpu or iloc_cpu are enabled.

If an EI level maskable interrupt is created during the interval from when either loc_cpu or iloc_cpu is issued until this service call is issued, the RI850V4 delays transition to the relevant interrupt processing (interrupt handler) until this service call is issued.

Note 1 This service call does not perform queuing of cancellation requests. If the system is in the CPU unlocked state, therefore, no processing is performed but it is not handled as an error.

Note 2 This service call manipulates PM*n* bits in the priority mask register (PMR) to disable acceptance of EI level maskable interrupts.
The PM*n* bits to be manipulated correspond to the interrupt priority range defined as the Maximum interrupt priority: maxintpri during configuration.
This service call does not manipulate the ID bit in the program status word (PSW).

Note 3 This service call does not cancel the dispatch disabled state that was set by issuing dis_dsp. If the system status before the CPU locked state is entered was the dispatch disabled state, the system status becomes the dispatch disabled state after this service call is issued.

Note 4 If a service call other than loc_cpu, iloc_cpu and sns_*xxx* is issued from when loc_cpu or iloc_cpu is issued until this service call is issued, the RI850V4 returns E_CTX.

Return value

Macro	Value	Description
E_OK	0	Normal completion.

sns_loc

Outline

Reference CPU state.

C format

```
BOOL    sns_loc (void);
```

Parameter(s)

None.

Explanation

This service call acquires the system status type when this service call is issued (CPU locked state or CPU unlocked state).

When this service call is terminated normally, the acquired system state type (TRUE: CPU locked state, FALSE: CPU unlocked state) is returned.

Return value

Macro	Value	Description
TRUE	1	Normal completion (CPU locked state).
FALSE	0	Normal completion (CPU unlocked state).

dis_dsp

Outline

Disable dispatching.

C format

```
ER      dis_dsp (void);
```

Parameter(s)

None.

Explanation

This service call changes the system status to the dispatch disabled state.

As a result, dispatch processing (task scheduling) is disabled from when this service call is issued until ena_dsp is issued.

If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when this service call is issued until ena_dsp is issued, the RI850V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until ena_dsp is issued, upon which the actual dispatch processing is performed in batch.

Note 1 The dispatch disabled state changed by issuing this service call must be cancelled before the task that issued this service call moves to the DORMANT state.

Note 2 This service call does not perform queuing of disable requests. If the system is in the dispatch disabled state, therefore, no processing is performed but it is not handled as an error.

Note 3 If a service call (such as wai_sem, wai_flg) that may move the status of an invoking task is issued from when this service call is issued until ena_dsp is issued, the RI850V4 returns E_CTX regardless of whether the required condition is immediately satisfied.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state.

ena_dsp

Outline

Enable dispatching.

C format

```
ER      ena_dsp (void);
```

Parameter(s)

None.

Explanation

This service call changes the system status to the dispatch enabled state.
As a result, dispatch processing (task scheduling) that has been disabled by issuing dis_dsp is enabled.
If a service call (chg_pri, sig_sem, etc.) accompanying dispatch processing is issued during the interval from when dis_dsp is issued until this service call is issued, the RI850V4 executes only processing such as queue manipulation, counter manipulation, etc., and the actual dispatch processing is delayed until this service call is issued, upon which the actual dispatch processing is performed in batch.

Note 1 This service call does not perform queuing of enable requests. If the system is in the dispatch enabled state, therefore, no processing is performed but it is not handled as an error.

Note 2 If a service call (such as wai_sem, wai_flg) that may move the status of an invoking task is issued from when dis_dsp is issued until this service call is issued, the RI850V4 returns E_CTX regardless of whether the required condition is immediately satisfied.

Return value

Macro	Value	Description
E_OK	0	Normal completion.
E_CTX	-25	Context error. - This service call was issued from a non-task. - This service call was issued in the CPU locked state.

sns_dsp

Outline

Reference dispatching state.

C format

```
BOOL      sns_dsp (void);
```

Parameter(s)

None.

Explanation

This service call acquires the system status type when this service call is issued (dispatch disabled state or dispatch enabled state).

When this service call is terminated normally, the acquired system state type (TRUE: dispatch disabled state, FALSE: dispatch enabled state) is returned.

Return value

Macro	Value	Description
TRUE	1	Normal completion (dispatching disabled state).
FALSE	0	Normal completion (dispatching enabled state).

sns_ctx

Outline

Reference contexts.

C format

```
BOOL    sns_ctx (void);
```

Parameter(s)

None.

Explanation

This service call acquires the context type of the processing program that issued this service call (non-task context or task context).

When this service call is terminated normally, the acquired context type (TRUE: non-task context, FALSE: task context) is returned.

Return value

Macro	Value	Description
TRUE	1	Normal completion (non-task contexts).
FALSE	0	Normal completion (task contexts).

sns_dpn

Outline

Reference dispatch pending state.

C format

```
BOOL    sns_dpn (void);
```

Parameter(s)

None.

Explanation

This service call acquires the system status type when this service call is issued (whether in dispatch pending state or not).

When this service call is terminated normally, the acquired system state type (TRUE: dispatch pending state, FALSE: dispatch not-pending state) is returned.

Return value

Macro	Value	Description
TRUE	1	Normal completion. (dispatch pending state)
FALSE	0	Normal completion. (any other states)

16.2.12 Service call management functions

The following shows the service calls provided by the RI850V4 as the service call management functions.

Table 16-12 Service Call Management Functions

Service Call	Function	Origin of Service Call
cal_svc	Invoke extended service call routine	Task, Non-task, Initialization routine
ical_svc	Invoke extended service call routine	Task, Non-task, Initialization routine

cal_svc
ical_svc

Outline

Invoke extended service call routine.

C format

```
ER_UINT cal_svc (FN fncd, VP_INT par1, VP_INT par2, VP_INT par3);
ER_UINT ical_svc (FN fncd, VP_INT par1, VP_INT par2, VP_INT par3);
```

Parameter(s)

I/O	Parameter	Description
I	FN *fncd*;	Function code of the extended service call routine to be invoked.
I	VP_INT *par1*;	The first parameter of the extended service call routine.
I	VP_INT *par2*;	The second parameter of the extended service call routine.
I	VP_INT *par3*;	The third parameter of the extended service call routine.

Explanation

These service calls call the extended service call routine specified by parameter *fncd*.

Note Extended service call routines that can be called using this service call are the routines whose transferred data total is less than four.

Return value

Macro	Value	Description
E_RSFN	-10	Invalid function code. - *fncd* ≦ 0x0 - *fncd* ＞ 0xff - Specified extended service call routine is not registered.
-	-	Normal completion (the extended service call routine's return value).

CHAPTER 17 SYSTEM CONFIGURATION FILE

This chapter explains the coding method of the system configuration file required to output information files (system information table file, system information header file and entry file) that contain data to be provided for the RI850V4.

17.1 Outline

The following shows the notation method of system configuration files.

- Character code
  Create the system configuration file using ASCII code.
  The CF850V4 distinguishes lower cases "a to z" and upper cases "A to Z".

  Note For Japanese language coding, Shit-JIS codes can be used only for comments.

- Comment
  In a system configuration file, parts between /* and */ and parts from two successive slashes (//) to the line end are regarded as comments.

- Numeric
  In a system configuration file, words starting with a numeric value (0 to 9) are regarded as numeric values.
  The CFV850V4 distinguishes numeric values as follows.

  Octal: Words starting with 0
  Decimal: Words starting with a value other than 0
  Hexadecimal: Words starting with 0x or 0X

  Note Unless specified otherwise, the range of values that can be specified as numeric values are limited from 0x0 to 0xffffffff.

- Symbol name
  In a system configuration file, words starting with an alphabetic character, "a to z, A to Z", or underscore "_" are regarded as symbol names.
  Describing a symbol name in the format "symbol name + offset" is also possible, but the offset must be a constant expression.
  The following shows examples of describing symbol names.
  The CF850V4 distinguishes between symbol names and other names based on the context in the system configuration file.

  [Correct]
```
func + 0x80000          // func name
symbol + 0x90 * 80      // symbol name
symbol + BASE           // data macro
```
  [Incorrect]
```
(func + 0x8000)         // The start character is illegal.
0x8000 + func           // The start character is illegal.
BASE + func             // Data macro BASE is handled as a symbol name.
func * 0x8000           // It is not the format of offset.
```
  Note Up to 4,095 characters can be specified for symbol names, including offset and spaces.

- Name
  In a system configuration file, words starting with an alphabetic character, "a to z, A to Z", or underscore "_" are regarded as names.
  The CF850V4 distinguishes between symbol names and other names based on the context in the system configuration file.

  Note Up to 255 characters can be specified for names.

- Preprocessing directives
  The following preprocessing directives can be coded in a system configuration file.

  #define, #elif, #else, #endif, #if, #ifdef, #ifndef, #include, #undef

- Keywords
  The words shown below are reserved by the CFV850V4 as keywords.
  Using these words for any other purpose specified is therefore prohibited.

  ATT_INI, CLK_INTNO, CPU_TYPE, CRE_CYC, CRE_DTQ, CRE_FLG, CRE_MBX, CRE_MPF, CRE_MPL, CRE_MTX, CRE_SEM, CRE_TSK, DEF_EXC, DEF_FPSR, DEF_INH, DEF_SVC, DEF_TEX, DEF_TIM, INCLUDE, INT_STK, MAX_CYC, MAX_DTQ, MAX_FLG, MAX_INT, MAX_INTPRI, MAX_MBX, MAX_MPF, MAX_MPL, MAX_MTX, MAX_PRI, MAX_SEM, MAX_SVC, MAX_TSK, MEM_AREA, NULL, SERVICECALL, RI_SERIES, SIZE_AUTO, STK_CHK, SYS_STK, TA_ACT, TA_ASM, TA_CLR, TA_DISINT, TA_DISPREEMPT, TA_ENAINT, TA_HLNG, TA_MFIFO, TA_MPRI, TA_OFF, TA_ON, TA_PHS, TA_RSTR, TA_STA, TA_TFIFO, TA_TPRI, TA_WMUL, TA_WSGL, TBIT_FLGPTN, TBIT_TEXPTN, TIC_DENO, TIC_NUME, TKERNEL_MAKER, TKERNEL_PRID, TKERNEL_PRVER, TKERNEL_SPVER, TMAX_ACTCNT, TMAX_MPRI, TMAX_SEMCNT, TMAX_SUSCNT, TMAX_TPRI, TMAX_WUPCNT, TMIN_MPRI, TMIN_TPRI, TSZ_DTQ, TSZ_MBF, TSZ_MPF, TSZ_MPL, TSZ_MPRIHD, VATT_IDL, VDEF_RTN, G3K, G3M, G3KH, G3MH

  Note In addition to the above words, service call names (such as act_tsk or slp_tsk), words starting with _kernel_, and the section names shown in Table B-1 to be used by the RI850V4 are reserved as keywords in the CF850V4.

17.2 Configuration Information

The configuration information that is described in a system configuration file is divided into the following three main types.

- Declarative Information
  Data related to a header file (header file name) in which data macro entities used in the system configuration file are defined.
  - Header file declaration
- System Information
  Data related to OS resources (such as real-time OS name, processor type) required for the RI850V4 to operate.
  - RI series information
  - Basic information
  - FPSR register information
  - Memory area information
- Static API Information
  Data related to management objects (such as task and task exception handling routine) used in the system.
  - Task information
  - Semaphore information
  - Eventflag information
  - Data queue information
  - Mailbox information
  - Mutex information
  - Fixed-sized memory pool information
  - Variable-sized memory pool information
  - Cyclic handler information
  - Interrupt handler information
  - Extended service call routine information
  - Initialization routine information
  - Idle routine information

17.2.1 Cautions

In the system configuration file, describe the system configuration information (Declarative Information, System Information, Static API Information) in the following order.

1) Declarative Information description

2) System Information description

3) Static API Information description

The information items in the System Information group (such as RI series information or Basic information) and those in the Static API Information group (such as Task information or Semaphore information) can be coded in any order within each respective group.

The following illustrates how the system configuration file is described.

Figure 17-1 System Configuration File Description Format

```
-- Declarative Information (Header file declaration) description
/* ......... */

-- System Information (RI series information, etc.) description
/* ......... */

-- Static API Information (Task information, etc.) description
/* ......... */
```

17.3 Declarative Information

The following describes the format that must be observed when describing the declarative information in the system configuration file.

The GOTHIC-FONT characters in following descriptions are the reserved words, and italic face characters are the portion that the user must write the relevant numeric value, symbol name, or keyword.

17.3.1 Header file declaration

The header file declaration defines File name: h_file.
The number of definable header file declaration items is not restricted.
The following shows the header file declaration format.

```
INCLUDE ("h_file");
```

The items constituting the header file declaration are as follows.

1) File name: *h_file*

Reflects the header file declaration specified in *h_file* into the system information header file output by the CF850V4.
As a result, macro definitions in *filename* can be referenced from a file in which the system information header file output by the CF850V4 is included.

Note If <sample.h> is specified in *h_file*, the header file definition (include processing) is output as:

```
#include        <sample.h>
```

If \"sample.h\" is specified in *h_file*, the header file definition (include processing) is output as:

```
#include        "sample.h"
```

to the system information header file.

17.4 System Information

The following describes the format that must be observed when describing the system information in the system configuration file.

The GOTHIC-FONT characters in following descriptions are the reserved words, and italic face characters are the portion that the user must write the relevant numeric value, symbol name, or keyword.

Items enclosed by square brackets "[ ]" can be omitted.

17.4.1 RI series information

The RI series information defines Real-time OS name: rtos_name, Version number: rtos_ver.
Only one information item can be defined as RI series information.
The following shows the RI series information format.

```
RI_SERIES (rtos_name, rtos_ver);
```

The items constituting the RI series information are as follows.

1) Real-time OS name: *rtos_name*

Specifies the real-time OS name.
The keyword that can be specified for *rtos_name* is the RI850V4.

2) Version number: *rtos_ver*

Specifies the version number for the RI850V4.
In RI_SERIES, values specifiable for *rtos_ver* is the “V2xy” (as version number).

Note If the version number of RI850V4 is "V2.01.23", *rtos_ver* should be set to "V201".

17.4.2 Basic information

The basic information defines CPU type: cpu, Base clock interval: tim_base, Base clock timer exception code: tim_intno, System stack size: sys_stksz, Whether to check stack: stkchk, Maximum priority: maxtpri, Maximum interrupt priority: maxintpri, Maximum number of interrupt handlers: maxint; Maximum value of exception code: maxintno.

Only one information item can be defined as basic information.

The following shows the basic information format.

```
[CPU_TYPE (chip_type);]
[DEF_TIM (tim_base);]
CLK_INTNO (tim_intno);
SYS_STK (sys_stksz);
[STK_CHK (stkchk);]
[MAX_PRI (maxtpri);]
[MAX_INTPRI (maxintpri);]
MAX_INT (maxint [, maxintno ] );
```

The items constituting the basic information are as follows.

1) CPU type: *cpu*

Specifies the type for a CPU.
If you are using the CS +, don't need to specify this specified item.
The keyword that can be specified for *chip_type* is G3K or G3M.

G3K: G3K core
G3M: G3M core

If omitted The CPU type should be the device type specified in the -cpu option. When the PE number is specified in the -peid option, the CPU type corresponding to the PE number should be specified. When the -cpu option setting is omitted, the CPU type should be G3K.

2) Base clock interval: *tim_base*

Specifies the base clock interval (unit:millisecond) of the timer to be used.
A value from 0x1 to 0xffff can be specified for *tim_base*.

If omitted "0x1ms" is specified as the base clock cycle of the RI850V4.

Note The base clock cycle means the occurrence interval of base clock timer interrupt *tim_intno*, which is required for implementing the TIME MANAGEMENT FUNCTIONS provided by the RI850V4. To initialize hardware used by the RI850V4 for time management (such as timers and controllers), the setting must therefore be made so as to generate base clock timer interrupts at the interval defined with *tim_base*.

3) Base clock timer exception code: *tim_intno*

Specifies the exception code for the base clock timer interrupt that is necessary to implement the time management facility provided by the RI850V4.
The value that can be specified for *tim_into* is an interrupt source name specified in the device file or a value from 0x1000 to the maximum exception code *maxintno*.

Note When an interrupt source name is specified for *tim_intno*, -cpu Δ *name* must be specified for the CF850V4 activation option.

4) System stack size: *sys_stksz*

Specifies the system stack size (in bytes).
A value from 0x0 to 0x7ffffffc (aligned to a 4-byte boundary) can be specified for *sys_stksz*.

Note 1 For expressions to calculate the system stack size, refer to "1) System stack".

Note 2 The memory area for system stack is secured from the ".kernel_work section".

Note 3 The stack size that is actually secured is calculated as the specified stack size plus "20 + 80 (size of context area of interrupt handler)".

5) Whether to check stack: *stkchk*

Specifies whether to check the stack overflows before the RI850V4 starts processing.
The keyword that can be specified for *flg* is TA_ON or TA_OFF.

TA_ON: Overflow is checked
TA_OFF: Overflow is not checked

Note Overflow is not checked by default.

6) Maximum priority: *maxtpri*

Specifies the maximum priority of the task.
A value from 0x1 to 0x20 can be specified for *maxtpri*.

If omitted "0x20" is specified as the maximum task priority.

7) Maximum interrupt priority: *maxintpri*

Specifies the maximum priority for EI level maskable interrupts to be managed by the RI850V4.
The following values can be specified for *maxintpri*.
When the CPU type of the target device is G3K: A value from INTPRI0 to INTPRI7.
When the CPU type of the target device is G3M: A value from INTPRI0 to INTPRI15.

Note 1 When INTPRI3 is specified, the RI850V4 manages interrupts within the range from priority INTPRI3 to the minimum interrupt priority.
The minimum interrupt priority is determined as follows. When the CPU type of the target device is G3K: IINTPRI7 is the minimum interrupt priority.
When the CPU type of the target device is G3M: INTPRI15 is the minimum interrupt priority.

Note 2 When the interrupt handlers for the EI level maskable interrupts are called in the reduced mode (the RINT bit in the reset vector base address (RBASE) or the exception handler vector address (EBASE) is set to 1), the maximum interrupt priority should be set to INTPRI0.

If omitted The maximum interrupt priority is set to INTPRI0.

8) Maximum number of interrupt handlers: *maxint*; Maximum value of exception code: *maxintno*

A value from 0x0 to 0x200 can be specified for *maxint*, and a value from 0x1000 to 0x11ff can be specified for *maxintno*.

Note 1 Specify for *maxint* “the total number of interrupt handlers defined in the Interrupt handler information”.

Note 2 When -cpu_*name* is specified as the CF850V4 activation option, the *maxintno* setting becomes invalid and the maximum exception code specified in the device file is used.

17.4.3 FPSR register information

The FPSR register information defines the following item.

1) FPSR register information: *fpsr*

The initial FPSR register value specified in this item is loaded in the FPSR register at the initial activation of a processing program (such as a task, a cyclic handler, or an interrupt handler).

The following shows the FPSR register information format.

```
[ DEF_FPSR ( fpsr ); ]
```

The item constituting the FPSR register information is shown below.

1) Initial FPSR register value: *fpsr*

Specifies the initial value to be loaded in the FPSR at initial activation of a processing program.

A value from 0x0 to 0xffffffff can be specified for *fpsr*.

Note that operation is not guaranteed if a value outside the range allowed by hardware is specified. See the hardware manual for the specific values.

If omitted The initial FPSR register value is "0x00020000".

Note 1 This item setting is only valid for a PE incorporating an FPU. If this item is specified for a PE that does not have an FPU, an error will occur.

Note 2 When using floating-point operation in the imprecise exception mode in a user routine, issue the syncp and synce instructions for synchronization to complete the floating-point operation before terminating the user routine processing by issuing a service call such as ext_tsk.

17.4.4 Memory area information

The memory area information defines Memory area name:sec_nam, Memory area size:memsz for a memory area.
The number of the definition as the memory area information is one, it's one per a section.
The following shows the memory area information format.

```
MEM_AREA (sec_nam, memsz);
```

The items constituting the memory area information are as follows.

1) Memory area name:*sec_nam*

Specifies the name of the memory area used for management objects.
Only the section-name (defined in link directive file) *.sec_nam* from which a dot is excluded can be specified for *sec_nam*.

2) Memory area size:*memsz*

Specifies the size of the memory area used for management objects (unit: bytes).
Only 4-byte boundary values from 0x0 to 0x7ffffffc, or "SIZE_AUTO" can be specified for *memsz*.

SIZE_AUTO: Total size of management objects defined in Basic information, Task information, etc.

Note For expressions to calculate the memory area size, refer to "APPENDIX B SIZE OF MEMORY".

Note When information regarding ".kernel_work" section has not been defined, the CF850V4 assumes that the following information is specified.

```
MEM_AREA ( kernel_work, SIZE_AUTO );
```

17.5 Static API Information

The following describes the format that must be observed when describing the static API information in the system configuration file.

The GOTHIC-FONT characters in following descriptions are the reserved words, and italic face characters are the portion that the user must write the relevant numeric value, symbol name, or keyword.

Items enclosed by square brackets "[ ]" can be omitted.

17.5.1 Task information

The task information defines ID number: tskid, Attribute: tskatr, Extended information: exinf, Start address: task, Initial priority: itskpri, Task stack size: stksz, memory area name: sec_nam, Reserved for future use: stk for a task.

The number of items that can be defined as task information is limited to one for each ID number.

The following shows the task information format.

sec_nam

```
CRE_TSK (tskid, { tskatr, exinf, task, itskpri, stksz[:sec_nam], stk });
```

The items constituting the task information are as follows.

1) ID number: *tskid*

Specifies the ID number for a task.
A value from 0x1 to 0xff, or a name, can be specified for *tskid*.

Note When a name is specified, the CF850V4 automatically assigns an ID number.
The CF850V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

```
#define tskid   value
```

2) Attribute: *tskatr*

Specifies the attribute for a task.
The keyword that can be specified for *tskatr* is TA_HLNG, TA_ASM, TA_ACT, TA_DISPREEMPT, TA_ENAINT and TA_DISINT.

[Coding language]
TA_HLNG: Start a task through a C language interface.
TA_ASM: Start a task through an assembly language interface.

[Initial activation state]
TA_ACT: Task is activated after the creation.

[Initial preemption state]
TA_DISPREEMPT: Preemption is disabled at task activation.

[Initial interrupt state]
TA_ENAINT: Acceptance of EI level maskable interrupts (from the Maximum interrupt priority: maxintpri to the minimum interrupt priority) is enabled.
TA_DISINT: Acceptance of EI level maskable interrupts (from the Maximum interrupt priority: maxintpri to the minimum interrupt priority) is disabled.

Note 1 If specification of TA_ACT is omitted, the DORMANT state is specified as the initial activation state.

Note 2 If specification of TA_DISPREEMPT is omitted, preempt acceptance is enabled.

Note 3 If specifications of TA_ENAINT and TA_DISINT are omitted, EI level maskable interrupts (from the Maximum interrupt priority: maxintpri to the minimum interrupt priority) are enabled in the initial state.

3) Extended information: *exinf*

Specifies the extended information for a task.
A value from 0x0 to 0xffffffff, or a symbol name, can be specified for *exinf*.

Note The target task can be manipulated by handling the extended information as if it were a function parameter.

4) Start address: *task*

Specifies the start address for a task.
A value from 0x0 to 0xfffffffe (aligned to a 2-byte boundary), or a symbol name, can be specified for *task*.

Note When a task is coded as follows, the symbol name specified for *task* should be func_task.

```
#include        <kernel.h>
#include        <kernel_id.h>

void
func_task ( VP_INT exinf )
{
        ............
        ............

        ext_tsk ( );
}
```

5) Initial priority: *itskpri*

Specifies the initial priority for a task.
A value from “0x1 to Maximum priority: maxtpri defined in the Basic information” can be specified for *itskpri*.

6) Task stack size: stksz, memory area name: *sec_nam*

Specifies the task stack size (unit: bytes) and the name of the memory area secured for the task stack.
Only 4-byte boundary values from 0x0 to 0x7ffffffc can be specified for *stksz*, and only memory area name *sec_nam* defined in Memory area information" can be specified for *sec_nam*.

Note 1 If specification of *sec_nam* is omitted, the task stack is allocated to “.kernel_work” section.

Note 2 The stack size that is actually secured is calculated as the specified stack size plus "*ctxsz* (size of context area of interrupt handler)". See 2）Task stack for details about *ctxsz*.

7) Reserved for future use: *stk*

System-reserved area.
Values that can be specified for *stk* are limited to NULL characters.

17.5.2 Semaphore information

The semaphore information defines ID number: semid, Attribute: sematr, Initial resource count: isemcnt, Maximum resource count: maxsem for a semaphore.

The number of items that can be defined as semaphore information is limited to one for each ID number.

The following shows the semaphore information format.

```
CRE_SEM (semid, { sematr, isemcnt, maxsem });
```

The items constituting the semaphore information are as follows.

1) ID number: *semid*

Specifies the ID number for a semaphore.
A value from 0x1 to 0xff, or a name, can be specified for *semid*.

Note When a name is specified, the CF850V4 automatically assigns an ID number.
The CF850V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

```
#define semid   value
```

2) Attribute: *sematr*

Specifies the task queuing method for a semaphore.
The keyword that can be specified for *sematr* is TA_TFIFO or TA_TPRI.

TA_TFIFO: Task wait queue is in FIFO order.
TA_TPRI: Task wait queue is in task priority order.

3) Initial resource count: *isemcnt*

Specifies the initial resource count for a semaphore.
A value from “0x0 to Maximum resource count: maxsem” can be specified for *isemcnt*.

4) Maximum resource count: *maxsem*

Specifies the maximum resource count for a semaphore.
A value from 0x1 to 0xffff can be specified for *maxsem*.

17.5.3 Eventflag information

The eventflag information defines ID number: flgid, Attribute: flgatr, Initial bit pattern: iflgptn for an eventflag.
The number of items that can be defined as eventflag information is limited to one for each ID number.
The following shows the eventflag information format.

```
CRE_FLG (flgid, { flgatr, iflgptn });
```

The items constituting the eventflag information are as follows.

1) ID number: *flgid*

Specifies the ID number for an eventflag.
A value from 0x1 to 0xff, or a name, can be specified for *flgid*.

Note When a name is specified, the CF850V4 automatically assigns an ID number.
The CF850V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

```
#define flgid   value
```

2) Attribute: *flgatr*

Specifies the attribute for an eventflag.
The keyword that can be specified for *flgatr* is TA_TFIFO, TA_TPRI, TA_WSGL, TA_WMUL and TA_CLR.

[Task queuing method]
TA_TFIFO: Task wait queue is in FIFO order.
TA_TPRI: Task wait queue is in task priority order.

[Queuing count]
TA_WSGL: Only one task is allowed to be in the WAITING state for the eventflag.
TA_WMUL: Multiple tasks are allowed to be in the WAITING state for the eventflag.

[Bit pattern clear]
TA_CLR: Bit pattern is cleared when a task is released from the WAITING state for eventflag.

Note 1 If specification of TA_TFIFO and TA_TPRI is omitted, tasks are queued in the order of bit pattern checking.

Note 2 If specification of TA_CLR is omitted, "not clear bit patterns if the required condition is satisfied" is set.

3) Initial bit pattern: *iflgptn*

Specifies the initial bit pattern for an eventflag.
A value from 0x0 to 0xffffffff can be specifies for *iflgptn*.

17.5.4 Data queue information

The data queue information defines ID number: dtqid, Attribute: dtqatr, Data count: dtqcnt, memory area name: sec_nam, Reserved for future use: dtq for a data queue.
The number of items that can be defined as data queue information is limited to one for each ID number.
The following shows the data queue information format.

```
CRE_DTQ (dtqid, { dtqatr, dtqcnt[:sec_nam], dtq });
```

The items constituting the data queue information are as follows.

1) ID number: *dtqid*

Specifies the ID number for a data queue.
A value from 0x1 to 0xff, or a name, can be specified for *dtqid*.

Note When a name is specified, the CF850V4 automatically assigns an ID number.
The CF850V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

```
#define dtqid   value
```

2) Attribute: *dtqatr*

Specifies the task queuing method for a data queue.
The keyword that can be specified for *dtqatr* is TA_TFIFO or TA_TPRI.

TA_TFIFO: Task wait queue is in FIFO order.
TA_TPRI: Task wait queue is in task priority order.

3) Data count: dtqcnt, memory area name: *sec_nam*

Specifies the maximum number of data units that can be queued to the data queue area of a data queue, and the name of the memory area secured for the data queue area.
Only values from 0x0 to 0xff can be specified for *dtqcnt*, and only memory area name *sec_nam* defined in Memory area information" can be specified for *sec_nam*.

Note If specification of *sec_nam* is omitted, the data queue is allocated to ".kernel_work" section.

4) Reserved for future use: *dtq*

System-reserved area.
Values that can be specified for *dtq* are limited to NULL characters.

17.5.5 Mailbox information

The mailbox information defines ID number: mbxid, Attribute: mbxatr, Maximum message priority: maxmpri, Reserved for future use: mprihd for a mailbox.
The number of items that can be defined as mailbox information is limited to one for each ID number.
The following shows the mailbox information format.

```
CRE_MBX (mbxid, { mbxatr, maxmpri, mprihd });
```

The items constituting the mailbox information are as follows.

1) ID number: *mbxid*

Specifies the ID number for a mailbox.
A value from 0x1 to 0xff, or a name, can be specified for *mbxid*.

Note When a name is specified, the CF850V4 automatically assigns an ID number.
The CF850V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

```
#define mbxid   value
```

2) Attribute: *mbxatr*

Specifies the attribute for a mailbox.
The keyword that can be specified for *mbxatr* is TA_TFIFO, TA_TPRI, TA_MFIFO and TA_MPRI.

[Task queuing method]
TA_TFIFO: Task wait queue is in FIFO order.
TA_TPRI: Task wait queue is in task priority order.

[Message queuing method]
TA_MFIFO: Message wait queue is in FIFO order.
TA_MPRI: Message wait queue is in message priority order.

3) Maximum message priority: *maxmpri*

Specifies the maximum message priority for a mailbox.
A value from 0x1 to 0x7fff can be specified for *maxmpri*.

Note maxmpri is valid only when TA_MPRI is specified for mqueue.
It is invalid when TA_MFIFO is specified for mqueue.

4) Reserved for future use: *mprihd*

System-reserved area.
Values that can be specified for *mprihd* are limited to NULL characters.

17.5.6 Mutex information

The mutex information defines ID number: mtxid, Attribute: mtxatr, Reserved for future use: ceilpri for a mutex.
The number of items that can be defined as mutex information is limited to one for each ID number.
The following shows the mutex information format.

```
CRE_MTX (mtxid, { mtxatr, ceilpri });
```

The items constituting the mutex information are as follows.

1) ID number: *mtxid*

Specifies the ID number for a mutex.
A value from 0x1 to 0xff, or a name, can be specified for *mtxid*.

Note When a name is specified, the CF850V4 automatically assigns an ID number.
The CF850V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

```
#define mtxid   value
```

2) Attribute: *mtxatr*

Specifies the task queuing method for a mutex.
The keyword that can be specified for *mtxatr* is TA_TFIFO or TA_TPRI.

TA_TFIFO: Task wait queue is in FIFO order.
TA_TPRI: Task wait queue is in task priority order.

3) Reserved for future use: *ceilpri*

System-reserved area.
Only values from "0x1 to maximum task priority *maxtpri* defined in Basic information" can be specified for *ceilpri*.

17.5.7 Fixed-sized memory pool information

The fixed-sized memory pool information defines ID number: mpfid, Attribute: mpfatr, Block count: blkcnt, Basic block size: blksz, memory area name: sec_nam, Reserved for future use: mpf for a fixed-sized memory pool.
The number of items that can be defined as fixed-sized memory pool information is limited to one for each ID number.
The following shows the fixed-sized memory pool information format.

```
CRE_MPF (mpfid, { mpfatr, blkcnt, blksz[:sec_nam], mpf });
```

The items constituting the fixed-sized memory pool information are as follows.

1) ID number: *mpfid*

Specifies the ID number for a fixed-sized memory pool.
A value from 0x1 to 0xff, or a name, can be specified for *mpfid*.

Note When a name is specified, the CF850V4 automatically assigns an ID number.
The CF850V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

```
#define mpfid   value
```

2) Attribute: *mpfatr*

Specifies the task queuing method for a fixed-sized memory pool.
The keyword that can be specified for *mpfatr* is TA_TFIFO or TA_TPRI.

TA_TFIFO: Task wait queue is in FIFO order.
TA_TPRI: Task wait queue is in task priority order.

3) Block count: *blkcnt*

Specifies the block count for a fixed-sized memory pool.
A value from 0x1 to 0x7fff can be specified for *blkcnt*.

4) Basic block size: *blksz*, memory area name: *sec_nam*

Specifies the size per block (unit: bytes) and the name of the memory area secured for the fixed-size memory pool.
Only 4-byte boundary values from 0x1 to 0x7ffffffc can be specified for *blksz*, and only memory area name *sec_area* defined in Memory area information" can be specified for *sec_nam*.

Note If specification of *sec_nam* is omitted, the fixed-sized memory pool is allocated to ".kernel_work" section.

5) Reserved for future use: *mpf*

System-reserved area.
Values that can be specified for *mpl* are limited to NULL characters.

17.5.8 Variable-sized memory pool information

The variable-sized memory pool information defines ID number: mplid, Attribute: mplatr, Pool size: mplsz, memory area name: sec_nam, Reserved for future use: mpl for a variable-sized memory pool.

The number of items that can be defined as variable-sized memory pool information is limited to one for each ID number.

The following shows the variable-sized memory pool information format.

```
CRE_MPL (mplid, { mplatr, mplsz[:sec_nam], mpl });
```

The items constituting the variable-sized memory pool information are as follows.

1) ID number: *mplid*

Specifies the ID number for a variable-sized memory pool.
A value from 0x1 to 0xff, or a name, can be specified for *mplid*.

Note When a name is specified, the CF850V4 automatically assigns an ID number.
The CF850V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

```
#define mplid   value
```

2) Attribute: *mplatr*

Specifies the task queuing method for a variable-sized memory pool.
The keyword that can be specified for *mplatr* is TA_TFIFO or TA_TPRI.

TA_TFIFO: Task wait queue is in FIFO order.
TA_TPRI: Task wait queue is in task priority order.

3) Pool size: *mplsz*, memory area name: *sec_nam*

Specifies the variable-size memory pool size (unit: bytes) and the name of the memory area secured for the variable-size memory pool.
Only 4-byte boundary values from 0x1 to 0x7ffffffc can be specified for *mplsz*, and only memory area name *sec_area* defined in Memory area information" can be specified for *sec_nam*.

Note If specification of *sec_nam* is omitted, the variable-sized memory pool is allocated to ".kernel_work" section.

4) Reserved for future use: *mpl*

System-reserved area.
Values that can be specified for *mpl* are limited to NULL characters.

17.5.9 Cyclic handler information

The cyclic handler information defines ID number: cycid, Attribute: cycatr, Extended information: exinf, Start address: cychdr, Activation cycle: cyctim, Activation phase: cycphs for a cyclic handler.
The number of items that can be defined as cycic handler information is limited to one for each ID number.
The following shows the cyclic handler information format.

```
CRE_CYC (cycid, { cycatr, exinf, cychdr, cyctim, cycphs });
```

The items constituting the cyclic handler information are as follows.

1) ID number: *cycid*

Specifies the ID number for a cyclic handler.
A value from 0x1 to 0xff, or a name, can be specified for *cycid*.

Note When a name is specified, the CF850V4 automatically assigns an ID number.
The CF850V4 outputs the relationship between a name and an ID number to the system information header file in the following format:

```
#define cycid   value
```

2) Attribute: *cycatr*

Specifies the attribute for a cyclic handler.
The keywords that can be specified for *cycatr* are TA_HLNG, TA_ASM, TA_STA and TA_PHS.

[Coding language]
TA_HLNG: Start a cyclic handler through a C language interface.
TA_ASM: Start a cyclic handler through an assembly language interface.

[Initial activation state]
TA_STA: Cyclic handlers is in an operational state after the creation.

[Activation phase]
TA_PHS: Cyclic handler is activated preserving the activation phase.

Note 1 If specification of TA_STA is omitted, the initial activation state is set to "non-operational state".

Note 2 If specification of TA_PHS is omitted, no activation phase items are saved.

3) Extended information: *exinf*

Specifies the extended information for a cyclic handler.
A value from 0x0 to 0xffffffff, or a symbol name, can be specified for *exinf*.

Note The target cyclic handler can be manipulated by handling the extended information as if it were a function parameter.

4) Start address: *cychdr*

Specifies the start address for a cyclic handler.
A value from 0x0 to 0xfffffffe (aligned to a 2-byte boundary), or a symbol name, can be specified for *cychdr*.

Note When a cyclic handler is coded as follows, the symbol name specified for *cychdr* should be func_cyc.

```
#include        <kernel.h>
#include        <kernel_id.h>
```

```
void
func_cyc ( VP_INT exinf )
{
        ............
        ............

        return;
}
```

5) Activation cycle: *cyctim*

Specifies the activation cycle (unit:millisecond) for a cyclic handler.
A value from 0x1 to 0x7fffffff (aligned to “clkcyc” multiple values) can be specified for *cyctim*.

Note If a value other than an integral multiple of the base clock cycle defined in Basic information is specified for *cyctim*, the CF850V4 assumes that an integral multiple is specified and performs processing.

6) Activation phase: *cycphs*

Specifies the activation phase (unit:millisecond) for a cyclic handler.
A value from 0x1 to 0x7fffffff (aligned to “clkcyc” multiple values) can be specified for *cycphs*.

Note 1 In the RI850V4, the initial activation phase means the relative interval from when generation of s cyclic handler is completed until the first activation request is issued.

Note 2 If a value other than an integral multiple of the base clock cycle defined in Basic information is specified for *cycphs*, the CF850V4 assumes that an integral multiple is specified and performs processing.

17.5.10 Interrupt handler information

The interrupt handler information defines Exception code: inhno, Attribute: inhatr, Start address: inthdr for an interrupt handler information.

The number of items that can be defined as interrupt handler information is limited to one for each exception code.

The following shows the interrupt handler information format.

```
DEF_INH (inhno, { inhatr, inthdr });
```

The items constituting the interrupt handler information are as follows.

1) Exception code: *inhno*

Specifies the exception code for an EI level maskable interrupt for which an interrupt handler is to be registered.
The value that can be specified for *inhno* is an interrupt source name specified in the device file or a value from 0x1000 to the maximum exception code specified in the Basic information.

Note When an interrupt source name is specified for *inhno*, -cpu_*name* must be specified for the CF850V4 activation option.

2) Attribute: *inhatr*

Specifies the language used to describe an interrupt handler.
The keyword that can be specified for *inhatr* is TA_HLNG or TA_ASM.

TA_HLNG: Start an interrupt handler through a C language interface.
TA_ASM: Start an interrupt handler through an assembly language interface.

3) Start address: *inthdr*

Specifies the start address for an interrupt handler.
A value from 0x0 to 0xfffffffe (aligned to a 2-byte boundary), or a symbol name, can be specified for *inthdr*.

Note When an interrupt handler is coded as follows, the symbol name specified for *inthdr* should be func_int.

```
#include        <kernel.h>
#include        <kernel_id.h>

void
func_int ( void )
{
        ............
        ............

        return;
}
```

17.5.11 Extended service call routine information

The extended service call routine information defines Function code: fncd, Attribute: svcatr, Start address: svcrtn for an extended service call routine.

The number of items that can be defined as extended service call routine information is limited to one for each function code.

The following shows the extended service call routine information format.

```
DEF_SVC (fncd, { svcatr, svcrtn });
```

The items constituting the extended service call routine information are as follows.

1) Function code: *fncd*

Specifies the function code for an extended service call routine.
A value from 0x1 to 0xff can be specified for *fncd*.

2) Attribute: *svcatr*

Specifies the language used to describe an extended service call routine.
The keyword that can be specified for *svcatr* is TA_HLNG or TA_ASM.

TA_HLNG:	Start an extended service call routine through a C language interface.
TA_ASM:	Start an extended service call routine through an assembly language interface.

3) Start address: *svcrtn*

Specifies the start address for an extended service call routine.
A value from 0x0 to 0xfffffffe (aligned to a 2-byte boundary), or a symbol name, can be specified for *svcrtn*.

Note When an extended service call routine handler is coded as follows, the symbol name specified for *svcrtn* should be func_svc.

```
#include        <kernel.h>
#include        <kernel_id.h>

ER_UINT
func_svc ( VP_INT par1, VP_INT par2, VP_INT par3 )
{
        ER_UINT ercd;

        ............
        ............

        return ( ercd );
}
```

17.5.12 Initialization routine information

The initialization routine information defines Attribute: iniatr, Extended information: exinf, Start address: inirtn for an initialization routine.

The number of initialization routine information items that can be specified is defined as being within the range of 0 to 254.

The following shows the idle initialization routine information format.

```
ATT_INI ({ initatr, exinf, inirtn });
```

The items constituting the initialization routine information are as follows.

1) Attribute: *iniatr*

Specifies the language used to describe an initialization routine.
The keyword that can be specified for *iniatr* is TA_HLNG or TA_ASM.

TA_HLNG:	Start an initialization routine through a C language interface.
TA_ASM:	Start an initialization routine through an assembly language interface.

2) Extended information: *exinf*

Specifies the extended information for an initialization routine.
A value from 0x0 to 0xffffffff, or a symbol name, can be specified for *exinf*.

Note The target initialization routine can be manipulated by handling the extended information as if it were a function parameter.

3) Start address: *inirtn*

Specifies the start address for an initialization routine.
A value from 0x0 to 0xfffffffe (aligned to a 2-byte boundary), or a symbol name, can be specified for *inirtn*.

Note When an initialization routine handler is coded as follows, the symbol name specified for *inirtn* should be func_ini.

```
#include        <kernel.h>

void
func_ini ( VP_INT exinf )
{
        ............
        ............

        return;
}
```

17.5.13 Idle routine information

The idle routine information defines Attribute: idlatr, Start address: idlrtn for an idle routine.
The number of idle routine information items that can be specified is defined as being within the range of 0 to 1.
The following shows the idle routine information format.

```
VATT_IDL ({ idlatr, idlrtn });
```

The items constituting the idle routine information are as follows.

1) Attribute: *idlatr*

Specifies the language used to describe an idle routine.
The keyword that can be specified for *idlatr* is TA_HLNG or TA_ASM.

TA_HLNG:	Start an idle routine through a C language interface.
TA_ASM:	Start an idle routine through an assembly language interface.

2) Start address: *idlrtn*

Specifies the start address for an idle routine.
A value from 0x0 to 0xfffffffe (aligned to a 2-byte boundary), or a symbol name, can be specified for *idlrtn*.

Note When an extended service call idle handler is coded as follows, the symbol name specified for *idlrtn* should be func_idl.

```
#include        <kernel.h>

void
func_idl ( void )
{
        ............
        ............

        return;
}
```

17.6 Description Examples

The following describes an example for coding the system configuration file.

Figure 17-2 Example of System Configuration File

```
-- Declarative Information description
INCLUDE (" \"kernel.h\" ");

-- System Information description
RI_SERIES (RI850V4, V201);

CPU_TYPE (G3M);
DEF_TIM (1);
CLK_INTNO (0x104c);
SYS_STK (0x800);
STK_CHK (TA_ON);
MAX_PRI (0x12);
MAX_INTPRI (INTPRI5);
MAX_INT (10, 0x1119);
DEF_FPSR ( 0x00020000 );

MEM_AREA (kernel_work, SIZE_AUTO);

-- Static API Information description
CRE_TSK ( ID_TASK1, { TA_HLNG | TA_ACT | TA_ENAINT, 0, task1, 1, 0x100, NULL } );
CRE_TSK ( ID_TASK2, { TA_HLNG | TA_ENAINT, 0, task2, 3, 0x50, NULL } );
CRE_TSK ( ID_TASK3, { TA_HLNG | TA_ENAINT, 0, task3, 3, 0x50, NULL } );
CRE_TSK ( ID_TASK4, { TA_HLNG | TA_ENAINT, 0, task4, 7, 0x50, NULL } );
CRE_TSK ( ID_TASK5, { TA_HLNG | TA_ENAINT, 0, task5, 5, 0x50, NULL } );

CRE_SEM ( ID_SEM1, { TA_TFIFO, 0x1, 0x1 } );

CRE_FLG ( ID_FLG1, { TA_TFIFO | TA_WMUL | TA_CLR, 0x0 } );

CRE_DTQ ( ID_DTQ1, { TA_TFIFO, 0x40, NULL } );

CRE_MBX ( ID_MBX1, { TA_TFIFO | TA_MFIFO, 0x10, NULL } );

CRE_MTX ( ID_MTX1, { TA_TFIFO, 0x10 } );

CRE_MPF ( ID_MPF1, { TA_TFIFO, 0x4, 0x10, NULL } );

CRE_MPL ( ID_MPL1, { TA_TFIFO, 0x50, NULL } );

CRE_CYC ( ID_CYC1, { TA_HLNG | TA_STA, 0x0, cychdr1, 1000, 5 } );

DEF_INH ( 0x1000, { TA_HLNG, inthdr1 } );
DEF_INH ( 0x1001, { TA_HLNG, inthdr2 } );

DEF_SVC ( 1, { TA_HLNG, svcrtn1 } );

ATT_INI ( { TA_HLNG, 0x0, inirtn } );

VATT_IDL ( { TA_HLNG, idlrtn } );
```

Note The RI850V4 provides sample source files for the system configuration file.

CHAPTER 18 CONFIGURATOR CF850V4

This chapter explains configurator CF850V4, which is provided by the RI850V4 as a utility tool useful for system construction.

18.1 Outline

To build systems (load module) that use functions provided by the RI850V4, the information storing data to be provided for the RI850V4 is required.

Since information files are basically enumerations of data, it is possible to describe them with various editors.

Information files, however, do not excel in descriptiveness and readability; therefore substantial time and effort are required when they are described.

To solve this problem, the RI850V4 provides a utility tool (configurator "CF850V4") that converts a system configuration file which excels in descriptiveness and readability into information files.

The CF850V4 reads the system configuration file as an input file, and then outputs information files.

The information files output from the CF850V4 are explained below.

- System information table file
  An information file that contains data related to OS resources (base clock interval, maximum priority, management object, or the like) required by the RI850V4 to operate.

- System information header file
  An information file that contains the correspondence between object names (task names, semaphore names, or the like) described in the system configuration file and IDs.

- Entry file
  A routine (Interrupt entry processing) dedicated to entry processing that holds processing to branch to the relevant processing (such as interrupt preprocessing "_kernel_int_entry") for the handler address to which the CPU forcibly passes control when an EI level maskable interrupt occurs.

18.2 Activation Method

18.2.1 Activating from command line

The following is how to activate the CF850V4 from the command line.

Note that, in the examples below, "C>" indicates the command prompt, "D" indicates pressing of the space key, and "<Enter>" indicates pressing of the enter key.

The activation options enclosed in "[ ]" can be omitted.

C> cf850v4.exe Δ [@<*command file*>] Δ [-peid=<*id*>] Δ [-cpu Δ <*name*>] Δ [-devpath=<*path*>] Δ [-i Δ <*SIT file*>] Δ [-e Δ <*Entry file*>] Δ [-d Δ <*Header file*>] Δ [-ni] Δ [-ne] Δ [-nd] Δ [-t Δ <*TOOL name*>] Δ [-T Δ <*Compiler path*>] Δ [-I Δ <*Include path*>] Δ [-np] Δ [-intbp=<*Interrupt Base Address*>] Δ [-ebase=<*Exception Base Address*>] Δ [-V] Δ [-help] Δ <*CF file*> [Enter]

The details of each activation option are explained below:

- @<*command_file*>
  Specifies the command file name to be input.

  If omitted The activation options specified on the command line is valid.

  Note 1 Specify the command file name <*command file*> within 255 characters including the path name.

  Note 2 When the command file name (including the path) includes a space, surround <*command file*> by double-quotation marks (").

  Note 3 For details about the command file, refer to "18.2.3 Command file".

- -peid=<*id*>
  Specifies the target PE number for which the application with RI850V4 is allocated.

  If omitted The CF850V4 performs processing with the assumption that -peid=1 is specified.

  Note 1 When the -cpu option is omitted, the CF850V4 ignores this activation option setting and outputs an information file for a single-core configuration.

  Note 2 A value from 1 to the maximum PE number in the target device can be specified for <*id*>.

- -cpu Δ <*name*>
  Specifies the device specification name for the target device (the character string of the device file name excluding the first character "d" and extension ".dvf").

  If omitted When the CC-RH compiler is used, -ne must be specified as the CF850V4 activation option.

  Note 1 When the device file name is dr7f701007.dvf, the character string specified for <*name*> should be r7f701007.

  Note 2 When -peid=<*id*> is specified, information regarding the specified PE number is read from the device file.

- -devpath=<*path*>
  Retrieves the device file corresponding to the target device specified with -cpu Δ <*name*> from the path folder.

  If omitted The device file is retrieved for the current folder.

  Note 1 Specify the search path <*path*> within 255 characters.

  Note 2 When the search path includes a space, surround <*path*> by double-quotation marks (").

- -i Δ <*SIT file*>
  Specify the output file name (system information table file name) while the CF850V4 is activated.

  If omitted The CF850V4 assumes that the following activation option is specified, and performs processing.

  REL Compiler (CC-RH) is used :-i Δ sit.s
  Green Hills Compiler (CCV850) is used :-i Δ sit.850

Note 1 Specify the output file name <*SIT file*> within 255 characters including the path name.

Note 2 When the output file name includes a space, surround <*SIT file*> by double-quotation marks (").

Note 3 If this activation option is specified together with -ni, the CF850V4 handles -ni as the valid option.

- -e Δ <*Entry file*>
Specify the output file name (entry file name) while the CF850V4 is activated.

If omitted The CF850V4 assumes that the following activation option is specified, and performs processing.

REL Compiler (CC-RH) is used :-e Δ entry.s
Green Hills Compiler (CCV850) is used :-e Δ entry.850

Note 1 Specify the output file name <*Entry file*> within 255 characters including the path name.

Note 2 When the output file name includes a space, surround <*Entry file*> by double-quotation marks (").

Note 3 If this activation option is specified together with -ne, the CF850V4 handles -ne as the valid option.

- -d Δ <*Header file*>
Specify the output file name (system information header file name) while the CF850V4 is activated.

If omitted If omitted The CF850V4 assumes that -d Δ kernel_id.h is specified and performs processing.

Note 1 Specify the output file name <*Header file*> within 255 characters including the path name.

Note 2 When the output file name includes a space, surround <*Header file*> by double-quotation marks (").

Note 3 If this activation option is specified together with -nd, the CF850V4 handles -nd as the valid option.

- -ni
Disables output of the system information table file.

If omitted The system information table file is output.

Note If this activation option is specified together with -i Δ <*SIT file*>, the CF850V4 handles this activation option as the valid option.

- -ne
Disables output of the entry file.

If omitted The entry file is output.

Note If this activation option is specified together with -e Δ <*Entry file*>, the CF850V4 handles this activation option as the valid option.

- -nd
Disables output of the system information header file.

If omitted If omitted The CF850V4 assumes that -d Δ kernel_id is specified and performs processing.

Note If this activation option is specified together with -d Δ <*Header file*>, the CF850V4 handles this activation option as the valid option.

- -t Δ <*TOOL name*>
Specifies the type of the C compiler package used.
Only REL and GHS can be specified for *tool* as the keyword.

If omitted The CF850V4 assumes that -t Δ REL is specified and performs processing.

- -T Δ <*Compiler path*>
Specifies the command search <*Compiler path*> folder for the C preprocessor of the C compiler package specified by -t Δ <*TOOL name*>.

If omitted The CF850V4 searches commands from a folder specified by environment variable (such as PATH).

Note 1 Specify the command search path name <*Compiler path*> within 255 characters.

Note 2 When the search path includes a space, surround <*Compiler path*> by double-quotation marks (").

- -I Δ <*Include path*>
  Specifies the command search <*Include path*> folder for <*Header file*> specified by Header file declaration.

  If omitted The CF850V4 starts searching from a folder where the input file specified by <*CF file*> is stored, the current folder, default search target folder of the C compiler package specified by -t Δ <*TOOL name*> in that order.

  Note 1 Specify the command search path name <*Include path*> within 255 characters.

  Note 2 When the search path includes a space, surround <*Include path*> by double-quotation marks (").

- -np
  Disables C preprocessor activation when the CF850V4 finished the analysis for syntax included in the system configuration file.

  If omitted The CF850V4 activates the C preprocessor of the C compiler package specified by -t Δ <*TOOL name*>.

- -intbp=<*Interrupt Base Address*>
  Specifies the base address of the interrupt handler address table, which is necessary when the entry file is output with the table reference method.

  If omitted If both this activation option and -ebase=<*Exception Base Address*> are omitted, the CF850V4 performs processing with the assumption that the direct vector method based on the reset vector address is selected as [Generate method] for the entry file.
  The reset vector address is set to the default value defined in the device file that is specified in -cpu Δ <*name*>. If the reset vector address value cannot be obtained from the device file, an error will occur.

  Note 1 A value from 0x200 to 0xfffff800 can be specified as the base address <*Interrupt Base Address*>.

  Note 2 If this activation option is specified together with -ebase=<*Exception Base Address*>, the CF850V4 handles this activation option as the valid option.

- -ebase=<*Exception Base Address*>
  Specifies the exception handler vector address, which is necessary when the entry file is output with the direct vector method.

  If omitted If both this activation option and -intbp=<*Interrupt Base Address*> are omitted, the CF850V4 performs processing with the assumption that the direct vector method based on the reset vector address is selected as [Generate method] for the entry file.
  The reset vector address is set to the default value defined in the device file that is specified in -cpu Δ <*name*>. If the reset vector address value cannot be obtained from the device file, an error will occur.

  Note 1 A value from 0x200 to 0xfffffe00 can be specified as the vector address <*Exception Base Address*>.

  Note 2 If this activation option is specified together with -intbp=<*Interrupt Base Address*>, the CF850V4 handles -intbp=<*Interrupt Base Address*> as the valid option.

- -V
  Outputs version information for the CF850V4 to the standard output.

  Note If this activation option is specified, the CF850V4 handles other activation options as invalid options and suppresses outputting of information files.

- -help
  Outputs the usage of the activation options for the CF850V4 to the standard output.

  Note If this activation option is specified, the CF850V4 handles other activation options as invalid options and suppresses outputting of information files.

- <*CF file*>
  Specifies the system configuration file name to be input.

  Note 1 Specify the input file name <*CF file*> within 255 characters including the path name.

  Note 2 When the input file name includes a space, surround <*CF file*> by double-quotation marks (").

  Note 3 This input file name can be omitted only when -V or -help is specified.

18.2.2 Activating from CS+

This is started when CS+ performs a build, in accordance with the setting on the Property panel, on the [System Configuration File Related Information] tab.

18.2.3 Command file

The CF850V4 performs command file support from the objectives that eliminate specified probable activation option character count restrictions in the command lines.

Description formats of the command file are described below.

1) Character code
Create a command file using ASCII code.

Note Shift-JIS and EUC-JP codes can be used only for comments.

2) Comment
A line beginning with # is handled as a comment.

3) Delimiter
A space, a tab, or a new-line character is handled as a delimiter.

4) Maximum number of lines
Up to 50 lines can be coded in a command file.

5) Maximum number of characters
Up to 16,384 characters per line can be coded in a command file.

An example of a command file is shown below.
In this example, the following activation options are included.

Target processor name:	r7f701z03
Device file search folder:	C:\CS+\CC\Device\RH850\Devicefile
System information table file name:	sit.s
Entry file name:	entry.s
System information header file name:	kernel_id.h
C compiler package type:	REL
Command search path for C compiler package:	C:\CS+\CC\CC-RH\V1.00.00\bin
Header file declaration search folder:	C:\tmp\inc850, and C:\Program Files\Sample\include
Vector address:	0x200
System configuration file name:	sys.cfg

Figure 18-1 Example of Command File Description

```
# Command File
-cpu rf701z03
-devpath=C:\CS+\CC\Device\RH850\Devicefile
-i sit.s
-e entry.s
-d kernel_id.h
-t REL
-T C:\CS+\CC\CC-RH\V1.00.00\bin
-I C:\tmp\inc850
-I “C:\Program Files\Sample\include“
-ebase=0x200
sys.cfg
```

18.2.4 Command input examples

The following shows CF850V4 command input examples.

In these examples, "C>" indicates the command prompt, "Δ" indicates the space key input, and "<Enter>" indicates the ENTER key input.

1) System configuration file sys.cfg is loaded from the current folder, the device file corresponding to the device specification name r7f701z03 is loaded from the C:\CS+\CC\Device\RH850\Devicefile folder as an input file, and system information table file sit.s, entry file entry.s for the direct vector method (vector address: 0x200), and system information header file kernel_id.h are then output. Command search processing for the preprocessor of the C compiler package from Renesas Electronics is done in the following order, and the relevant preprocessor is activated when the CF850V4 has finished the analysis for the syntax of the system configuration file.

1. C:\CS+\CC\CC-RH\V1.00.00\bin
2. Folders defined by environment variables (such as PATH)

File search processing for the header files specified in the header file information is performed in the following order.

1. C:\tmp\inc850
2. C:\Program Files\Sample\include

```
C> cf850v4 Δ -cpu Δ rf701z03 Δ -devpath=C:\CS+\CC\Device\RH850\Devicefile Δ -i Δ sit.s Δ -e Δ entry.s Δ
   -d Δ kernel_id.h Δ -t Δ REL Δ -T Δ C:\CS+\CC\CC-RH\V1.00.00\bin Δ -I Δ C:\tmp\inc850 Δ -I Δ
   "C:\Program Files\Sample\include" Δ -ebase=0x200 Δ sys.cfg［Enter］
```

2) CF850V4 version information is output to the standard output.

```
C> cf850v4 Δ -V [Enter]
```

3) Information related to the CF850V4 activation option (type, usage, or the like) is output to the standard output.

```
C> cf850v4 Δ -help [Enter]
```

APPENDIX A WINDOW REFERENCE

This appendix explains the window/panels that are used when the activation option for the CF850V4 is specified from the integrated development environment platform CS+.

A.1 Description

The following shows the list of window/panels.

Table A-1 List of Window/Panels

Window/Panel Name	Function Description
Main window	This is the first window to be opened when CS+ is launched. This window is used to manipulate the CS+ components (such as the build tool and resource information tool).
Project Tree panel	This panel is used to display the project components (such as the microcontroller and build tool) in tree view.
Property panel	This panel is used to display information regarding the node selected on the Project Tree panel and change the settings of the information.

Main window

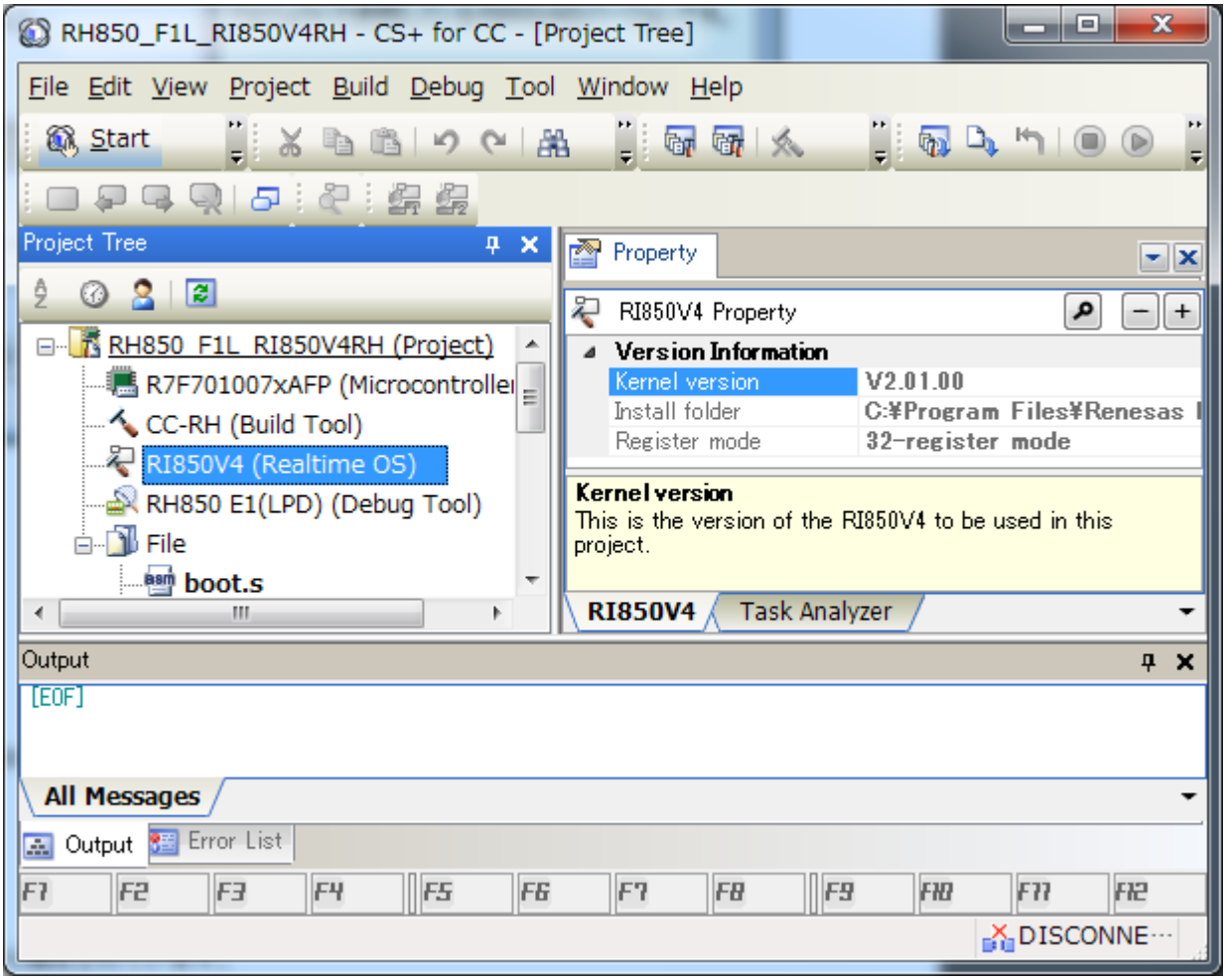
Outline

This is the first window to be opened when CS+ is launched. This window is used to manipulate the CS+ components (such as the build tool and resource information tool).

This window can be opened as follows:

- Select Windows [start] -> [All programs] -> [Renesas Electronics CS+] -> [CS+ for CC (RL78, RX, RH850)]

Display image



Explanation of each area

1) Menu bar

This area contains the following group of menus.

- [View] menu

Realtime OS	The [View] menu shows the cascading menu to start the tools of Real-Time OS.
Resource Information	Opens the Realtime OS Resource Information panel. Note that this menu is disabled when the debug tool is not connected.
Performance Analyzer	This menu is always disabled.
Task Analyzer 1	Opens the Realtime OS Task Analyzer 1 panel. Note that this menu is disabled when the debug tool is not connected.
Task Analyzer 2	Opens the Realtime OS Task Analyzer 2 panel. Note that this menu is disabled when the debug tool is not connected.

2) Toolbar

Displays the buttons relate to Realtime OS.
Buttons on the toolbar can be customized in the User Setting dialog box. You can also create a new toolbar in the same dialog box.

- Realtime OS toolbar

	Opens the Realtime OS Resource Information panel. Note that this button is disabled when the debug tool is not connected.
	Opens the Realtime OS Task Analyzer 1 panel. Note that this menu is disabled when the debug tool is not connected.
	Opens the Realtime OS Task Analyzer 2 panel. Note that this menu is disabled when the debug tool is not connected.

3) Panel display area

The following panels are displayed in this area.

- Project Tree panel
- Property panel
- Output panel

See the each panel section for details of the contents of the display.

Note See CS+ Integrated Development Environment User's Manual: Build for details about the Output panel.

Project Tree panel

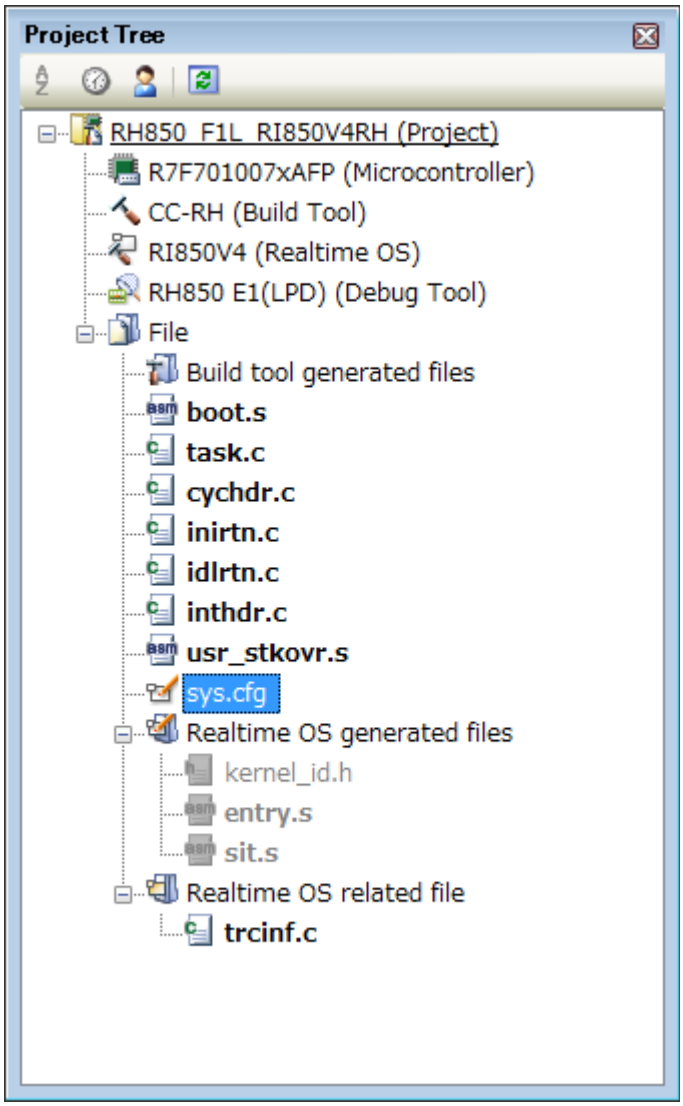
Outline

This panel is used to display the project components (such as the microcontroller and build tool) in tree view.

This panel can be opened as follows:

- Select [View] menu -> [Project Tree]

Display image



Explanation of each area

1) Project tree area

Project components are displayed in tree view with the following given node.

Node	Description
RI850V4 (Realtime OS) (referred to as “Realtime OS node”)	Realtime OS to be used.
xxx.cfg	System configuration file.
Realtime OS generated files (referred to as “Realtime OS generated files node”)	The following information files appear directly below the node created when a system configuration file is added. - System information table file (.s) - System information header file (.h) - Entry file (.s) This node and files displayed under this node cannot be deleted directly. This node and files displayed under this node will no longer appear if you remove the system configuration file from the project.
Realtime OS related file (referred to as “Realtime OS related files node”)	The following information file appears directly below the node. - Trace information file (trcinf.c) This node and the file displayed under this node cannot be deleted.

Property panel

Outline

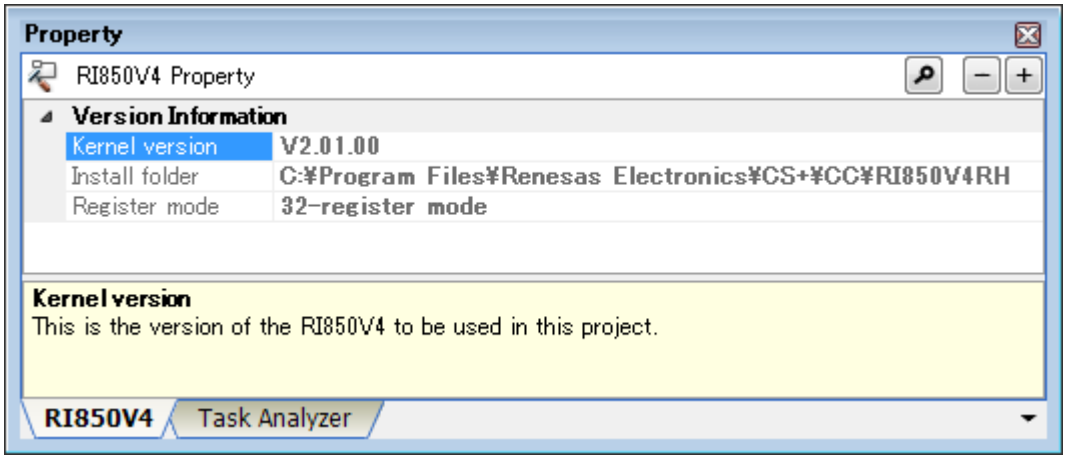
This panel is used to display information regarding the node selected on the Project Tree panel and change the settings of the information.

This panel can be opened as follows:

- On the Project Tree panel, select a component such as the Realtime OS node or the system configuration file, and then select the [View] menu -> Property] or select [Property] from the context menu.

Note When the Property panel is already open, selecting a component such as the Realtime OS node or the system configuration file on the Project Tree panel displays the detailed information regarding the selected component.

Display image



Explanation of each area

1) Selected node area

Display the name of the selected node on the Project Tree panel.
When multiple nodes are selected, this area is blank.

2) Detailed information display/change area

In this area, the detailed information on the Realtime OS node, system configuration file, or the like that is selected on the Project Tree panel is displayed by every category in the list. And the settings of the information can be changed directly.

Mark ⊟ indicates that all the items in the category are expanded. Mark ⊞ indicates that all the items are collapsed. You can expand/collapse the items by clicking these marks or double clicking the category name.
See the section on each tab for the details of the display/setting in the category and its contents.

3) Property description area

Display the brief description of the categories and their contents selected in the detailed information display/change area.

4) Tab selection area

Categories for the display of the detailed information are changed by selecting a tab.
In this panel, the following tabs are contained (see the section on each tab for the details of the display/setting on the tab).

- When the Realtime OS node is selected on the Project Tree panel
  - [RI850V4] tab
- When the system configuration file is selected on the Project Tree panel
  - [System Configuration File Related Information] tab
  - [File Information] tab
- When the Realtime OS generated files node is selected on the Project Tree panel
  - [Category Information] tab
- When the system information table file or entry file is selected on the Project Tree panel
  - [Build Settings] tab
  - [Individual Assemble Options] tab
  - [File Information] tab
- When the system information header file is selected on the Project Tree panel
  - [File Information] tab
- When the trace information file is selected on the Project Tree panel
  - [Build Settings] tab
  - [Individual Assemble Options] tab
  - [File Information] tab

Note1 See "CS+ Integrated Development Environment User's Manual: CC-RH Build Tool Operation" for details about the [File Information], [Category Information], [Build Settings], [Individual Assemble Options], and [Individual Compile Options] tabbed pages.

Note2 When multiple components are selected on the Project Tree panel, only the tab that is common to all the components is displayed. If the value of the property is modified, that is taken effect to the selected components all of which are common to all.

[RI850V4] tab

Outline

This tab shows the detailed information on RI850V4 to be used categorized by the following.

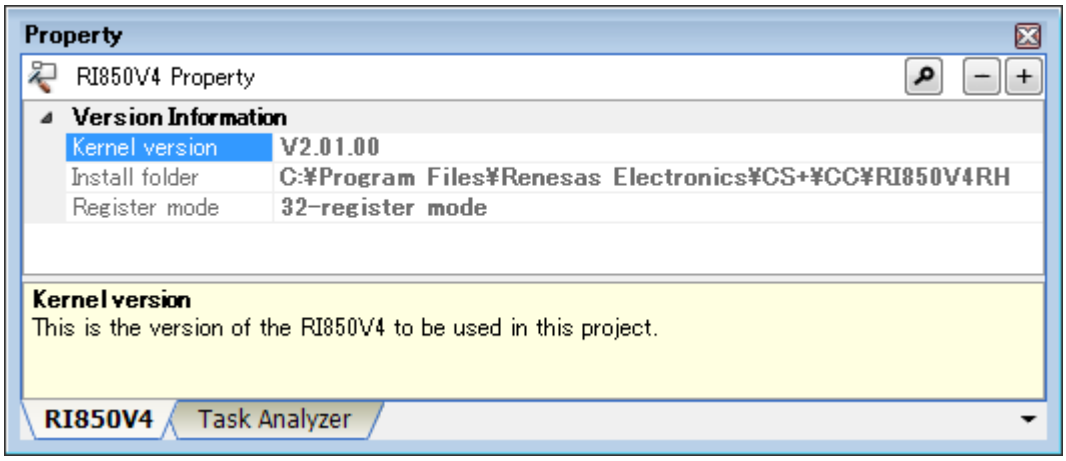
- Version Information

This tab can be opened as follows:

- On the Project Tree panel, select a component such as the Realtime OS node or the system configuration file, and then select the [View] menu -> Property] or select [Property] from the context menu.

Note When the Property panel is already open, selecting a component such as the Realtime OS node or the system configuration file on the Project Tree panel displays the detailed information regarding the selected component.

Display image



Explanation of each area

1) [Version Information]

The detailed information on the version of the RI850V4 are displayed.

Kernel version	Display the version of RI850V4 to be used.	
	Default	*The latest version of the installed RI850V4 package*
	How to change	Changes not allowed
Install folder	Display the folder in which RI850V4 to be used is installed with the absolute path.	
	Default	*The folder in which RI850V4 to be used is installed*
	How to change	Changes not allowed

Register mode	Display the register mode set in the project. Display the same value as the value of the [Select register mode] property of the build tool.	
	Default	*The register mode selected in the property of the build tool*
	How to change	Changes not allowed

[Task Analyzer] tab

Outline

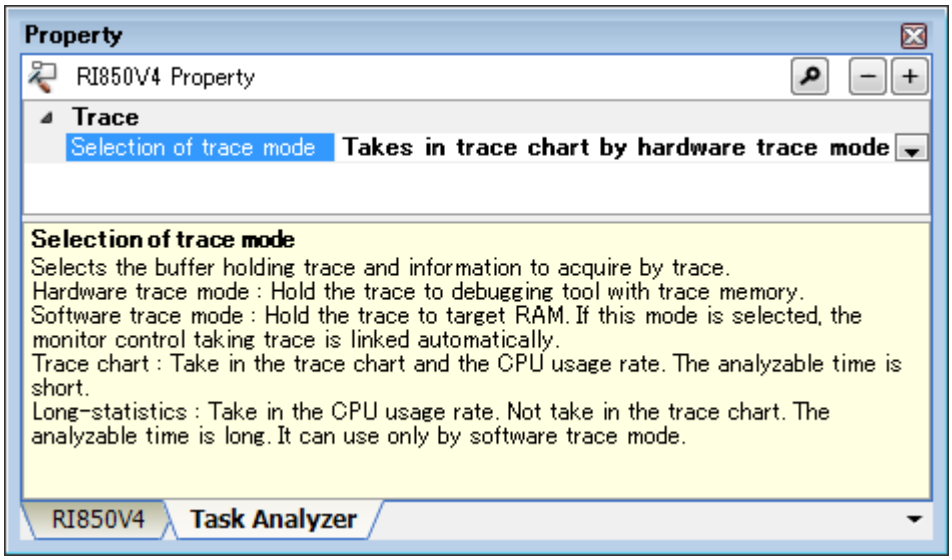
This page is used to display and change the settings of various information required when using the task analyzer tool, which is a utility tool provided by the RI850V4, to analyze the history (trace data) of processing program execution.

This tab can be opened as follows:

- On the Project Tree panel, select a component such as the Realtime OS node or the system configuration file, and then select the [View] menu -> Property] or select [Property] from the context menu.

Note When the Property panel is already open, selecting a component such as the Realtime OS node or the system configuration file on the Project Tree panel displays the detailed information regarding the selected component.

Display image


Property
RI850V4 Property
Trace
Selection of trace mode
Takes in trace chart by hardware trace mode
Selection of trace mode
Selects the buffer holding trace and information to acquire by trace.
Hardware trace mode : Hold the trace to debugging tool with trace memory.
Software trace mode : Hold the trace to target RAM. If this mode is selected, the monitor control taking trace is linked automatically.
Trace chart : Take in the trace chart and the CPU usage rate. The analyzable time is short.
Long-statistics : Take in the CPU usage rate. Not take in the trace chart. The analyzable time is long. It can use only by software trace mode.
RI850V4
Task Analyzer

Explanation of each area

1) [Trace] category

Displays and changes the settings of various information required when using the utility tool "task analyzer tool" to analyze the history (trace data) of processing program execution.

<table>
<tr><td rowspan="7">Selection of trace mode</td><td colspan="3">Select the type of information to be acquired as trace data and the location where trace data is to be stored.</td></tr>
<tr><td>Default</td><td colspan="2">Not tracing</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list</td></tr>
<tr><td rowspan="4">Restriction</td><td>Not tracing</td><td>Does not use the task analyzer tool.</td></tr>
<tr><td>Taking in trace chart by hardware trace mode</td><td>Acquires information in a trace chart (such as the execution transition state of the processing program and the state of Realtime OS resource usage) and CPU usage status as trace data.
The trace buffer is allocated in the trace memory prepared by the debug tool.</td></tr>
<tr><td>Taking in trace chart by software trace mode</td><td>Acquires information in a trace chart (such as the execution transition state of the processing program and the state of Realtime OS resource usage) and CPU usage status as trace data.
The trace buffer is allocated in the area selected in [Select the buffer].</td></tr>
<tr><td>Taking in long-statistics by software trace mode</td><td>Acquires the CPU usage status as trace data.
The trace buffer is allocated in the prespecified section “.kernel_data_trace.bss”.</td></tr>
<tr><td rowspan="5">Operation after used up the buffers</td><td colspan="3">Select the operation after using up the trace buffer.
This item is displayed only when "Taking in trace chart by software trace mode" is selected in [Selection of trace mode].</td></tr>
<tr><td>Default</td><td colspan="2">Continue to execution while the buffers overwriting</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="2">Restriction</td><td>Continue to execution while the buffers overwriting</td><td>Overwrites the oldest trace data written to the buffer.</td></tr>
<tr><td>Stop the trace taking in</td><td>Stops writing to the trace buffer.</td></tr>
<tr><td rowspan="4">Buffer size</td><td colspan="3">Specify the size of the trace buffer (bytes).
This item is displayed only when "Taking in trace chart by software trace mode" is selected in [Selection of trace mode].</td></tr>
<tr><td>Default</td><td colspan="2">0x100</td></tr>
<tr><td>How to change</td><td colspan="2">Enter directly in the text box.</td></tr>
<tr><td>Restriction</td><td colspan="2">0x10 - 0xfffffffc</td></tr>
</table>

<table>
<tr><td rowspan="5">Select the buffer</td><td colspan="3">Select the location to store trace data.
This item is displayed only when "Taking in trace chart by software trace mode" is selected in [Selection of trace mode].</td></tr>
<tr><td>Default</td><td colspan="2">Kernel buffer</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="2">Restriction</td><td>Kernel buffer</td><td>Allocates the trace buffer in the prespecified section “.kernel_data_trace.bss”.</td></tr>
<tr><td>Another buffer</td><td>Allocates the trace buffer at the address specified in [Buffer address].</td></tr>
<tr><td rowspan="4">Buffer address</td><td colspan="3">Specify the start address of the area to be allocated as the trace buffer.
This item is displayed only when "Another buffer" is selected in [Select the buffer].</td></tr>
<tr><td>Default</td><td colspan="2">0x100</td></tr>
<tr><td>How to change</td><td colspan="2">Enter directly in the text box.</td></tr>
<tr><td>Restriction</td><td colspan="2">0x10 - 0xfffffff0</td></tr>
</table>

[System Configuration File Related Information] tab

Outline

This tab shows the detailed information on the using system configuration file categorized by the following and the configuration can be changed.

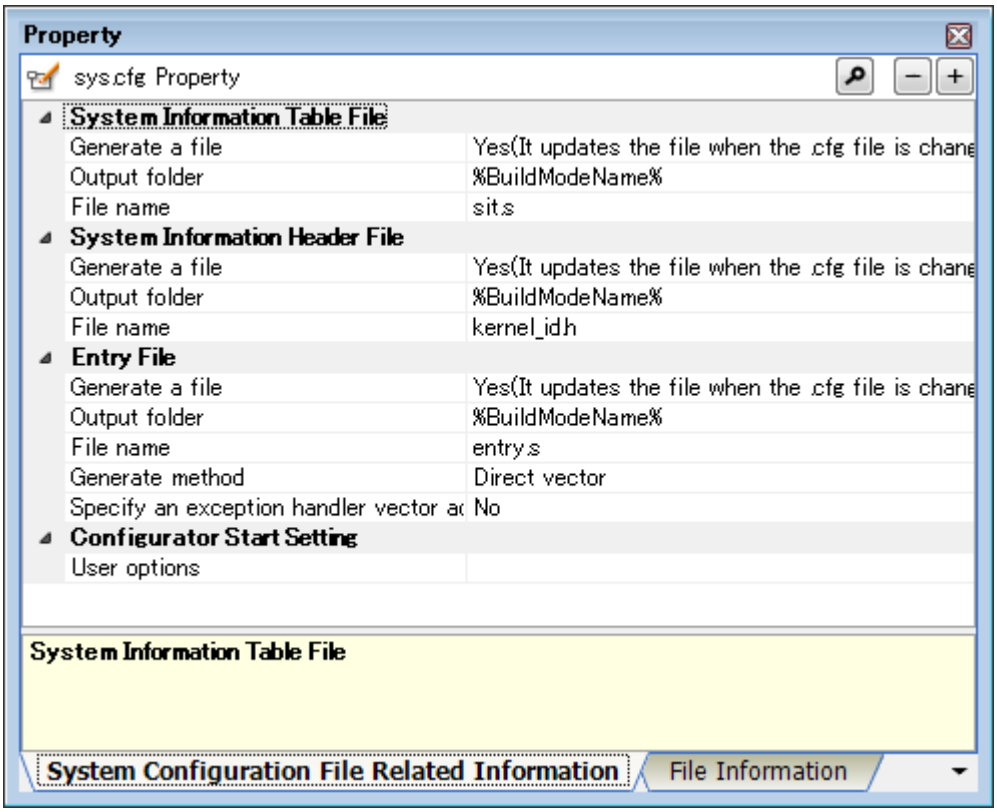
- System information table file
- System information header file
- Entry file

This tab can be opened as follows:

- On the Project Tree panel, select a component such as the Realtime OS node or the system configuration file, and then select the [View] menu -> Property] or select [Property] from the context menu.

Note When the Property panel is already open, selecting a component such as the Realtime OS node or the system configuration file on the Project Tree panel displays the detailed information regarding the selected component.

Display image



Explanation of each area

1) [System Information Table File]

The detailed information on the system information table file are displayed and the configuration can be changed.

<table>
<tr><td rowspan="6">Generate a file</td><td colspan="3">Select whether to generate a system information table file and whether to update the file when the system configuration file is changed.</td></tr>
<tr><td>Default</td><td colspan="2">Yes(It updates the file when the .cfg file is changed)(-i)</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="3">Restriction</td><td>Yes(It updates the file when the .cfg file is changed)(-i)</td><td>Generates a new system information table file and displays it on the project tree.
If the system configuration file is changed when there is already a system information table file, then the system information table file is updated.</td></tr>
<tr><td>Yes(It does not update the file when the .cfg file is changed)(-ni)</td><td>Does not update the system information table file when the system configuration file is changed.
An error occurs during build if this item is selected when the system information table file does not exist.</td></tr>
<tr><td>No(It does not register the file to the project)(-ni)</td><td>Does not generate a system information table file and does not display it on the project tree.
If this item is selected when there is already a system information table file, then the file itself is not deleted.</td></tr>
<tr><td rowspan="4">Output folder</td><td colspan="3">Specify the folder for outputting the system information table file.
If a relative path is specified, the reference point of the path is the project folder.
If an absolute path is specified, the reference point of the path is the project folder (unless the drives are different).
The following macro name is available as an embedded macro.
%BuildModeName%: Replaces with the build mode name.
If this field is left blank, macro name "%BuildModeName%" will be displayed.
This property is not displayed when [No(It does not register the file that is added to the project)(-ni)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td colspan="2">%BuildModeName%</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box or edit by the Browse For Folder dialog box which appears when clicking the [...] button.</td></tr>
<tr><td>Restriction</td><td colspan="2">Up to 247 characters</td></tr>
<tr><td rowspan="4">File name</td><td colspan="3">Specify the system information table file name.
If the file name is changed, the name of the file displayed on the project tree.
Use the extension ".s". If the extension is different or omitted, ".s" is automatically added.
You cannot specify the same file name as the value of the [File name] property in the [Entry File] category.
This property is not displayed when [No(It does not register the file that is added to the project)(-ni)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td colspan="2">sit.s</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box.</td></tr>
<tr><td>Restriction</td><td colspan="2">Up to 259 characters</td></tr>
</table>

2) [System Information Header File]

The detailed information on the system information header file are displayed and the configuration can be changed.

<table>
<tr><td rowspan="6">Generate a file</td><td colspan="3">Select whether to generate a system information header file and whether to update the file when the system configuration file is changed.</td></tr>
<tr><td>Default</td><td colspan="2">Yes(It updates the file when the .cfg file is changed)(-d)</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="3">Restriction</td><td>Yes(It updates the file when the .cfg file is changed)(-d)</td><td>Generates a system information header file and displays it on the project tree.
If the system configuration file is changed when there is already a system information header file, then the system information header file is updated.</td></tr>
<tr><td>Yes(It does not update the file when the .cfg file is changed)(-nd)</td><td>Does not update the system information header file when the system configuration file is changed.
An error occurs during build if this item is selected when the system information header file does not exist.</td></tr>
<tr><td>No(It does not register the file to the project)(-nd)</td><td>Does not generate a system information header file and does not display it on the project tree.
If this item is selected when there is already a system information header file, then the file itself is not deleted.</td></tr>
<tr><td rowspan="4">Output folder</td><td colspan="3">Specify the folder for outputting the system information header file.
If a relative path is specified, the reference point of the path is the project folder.
If an absolute path is specified, the reference point of the path is the project folder (unless the drives are different).
The following macro name is available as an embedded macro.
%BuildModeName%: Replaces with the build mode name.
If this field is left blank, macro name "%BuildModeName%" will be displayed.
This property is not displayed when [No(It does not register the file that is added to the project)(-nd)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td colspan="2">%BuildModeName%</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box or edit by the Browse For Folder dialog box which appears when clicking the [...] button.</td></tr>
<tr><td>Restriction</td><td colspan="2">Up to 247 characters</td></tr>
<tr><td rowspan="4">File name</td><td colspan="3">Specify the system information header file name.
If the file name is changed, the name of the file displayed on the project tree.
Use the extension ".h". If the extension is different or omitted, ".h" is automatically added.
This property is not displayed when [No(It does not register the file that is added to the project)(-nd)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td colspan="2">kernel_id.h</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box.</td></tr>
<tr><td>Restriction</td><td colspan="2">Up to 259 characters</td></tr>
</table>

3) [Entry File]

The detailed information on the entry file are displayed and the configuration can be changed.

<table>
<tr><td rowspan="5">Generate a file</td><td colspan="3">Select whether to generate an entry file and whether to update the file when the system configuration file is changed.</td></tr>
<tr><td>Default</td><td colspan="2">Yes(It updates the file when the .cfg file is changed)(-e)</td></tr>
<tr><td>How to change</td><td colspan="2">Select from the drop-down list.</td></tr>
<tr><td rowspan="3">Restriction</td><td>Yes(It updates the file when the .cfg file is changed)(-e)</td><td>Generates an entry file and displays it on the project tree.
If the system configuration file is changed when there is already an entry file, then the entry file is updated.</td></tr>
<tr><td>Yes(It does not update the file when the .cfg file is changed)(-ne)</td><td>Does not update the entry file when the system configuration file is changed.
An error occurs during build if this item is selected when the entry file does not exist.</td></tr>
<tr><td>No(It does not register the file to the project)(-ne)</td><td>Does not generate an entry file and does not display it on the project tree.
If this item is selected when there is already an entry file, then the file itself is not deleted.</td></tr>
<tr><td rowspan="4">Output folder</td><td colspan="3">Specify the folder for outputting the entry file.
If a relative path is specified, the reference point of the path is the project folder.
If an absolute path is specified, the reference point of the path is the project folder (unless the drives are different).
The following macro name is available as an embedded macro.
%BuildModeName%: Replaces with the build mode name.
If this field is left blank, macro name "%BuildModeName%" will be displayed.
This property is not displayed when [No(It does not register the file that is added to the project)(-ne)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td colspan="2">%BuildModeName%</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box or edit by the Browse For Folder dialog box which appears when clicking the [...] button.</td></tr>
<tr><td>Restriction</td><td colspan="2">Up to 247 characters</td></tr>
<tr><td rowspan="4">File name</td><td colspan="3">Specify the entry file.
If the file name is changed, the name of the file displayed on the project tree.
Use the extension ".s". If the extension is different or omitted, ".s" is automatically added.
You cannot specify the same file name as the value of the [File name] property in the [System Information Table File] category.
This property is not displayed when [No(It does not register the file that is added to the project)(-ne)] in the [Generate a file] property is selected.</td></tr>
<tr><td>Default</td><td colspan="2">entry.s</td></tr>
<tr><td>How to change</td><td colspan="2">Directly enter to the text box.</td></tr>
<tr><td>Restriction</td><td colspan="2">Up to 259 characters</td></tr>
</table>

Generate method	Select the branch method when a base clock timer interrupt defined in the Basic information or an EI level maskable interrupt defined in the Interrupt handler information occurs.		
	Default	Direct vector	
	How to change	Select from the drop-down list.	
	Restriction	Table reference (-intbp)	Generates an entry file for the table reference method.
		Direct vector	Generates an entry file for the direct vector method.
Base address of the interrupt handler address table	Specifies the base address of the interrupt handler address table. This item is not displayed when "Direct vector" is selected in [Generate method].		
	Default	0x0	
	How to change	Enter directly in the text box.	
	Restriction	0x0 - 0xffffffff	
Specify the exception handler vector address	Select whether to specify the exception handler vector address.		
	Default	No	
	How to change	Select from the drop-down list.	
	Restriction	Yes (-ebase)	Specifies the exception handler vector address.
		No	Does not specify the exception handler vector address.
Exception handler vector address	Specify the exception handler vector address. This item is not displayed when "No" is selected in [Specify the exception handler vector address].		
	Default	0x0	
	How to change	Enter directly in the text box.	
	Restriction	0x0 - 0xffffffff	

4) [Configurator Start Setting] category]

The activation option for the CF850V4 configurator can be specified.

User options	Specify the desired user option to be passed to CF850V4.		
	Default	-	
	How to change	Enter directly in the text box.	
	Restriction	-peid= *value*	Specifies a PE number. For details of value setting, see "18.2.1 Activating from command line".

Note [Configurator Start Setting] should be specified for each project that uses the RI850V4 and multiple PE numbers cannot be specified as the user option at the same time.
Therefore, only a single PE should be handled in each project that uses the RI850V4.

APPENDIX B SIZE OF MEMORY

This appendix explains the size of the memory area.

B.1 Description

The memory areas used and managed by the RI850V4 are divided into eight sections by use.

Table B-1 Memory Area

Section Name	Outline
.kernel_system	Area where executable code of RI850V4 is allocated.
.kernel_const	Area where static data of RI850V4 is allocated.
.kernel_data	Area where dynamic data of RI850V4 is allocated.
.kernel_data_init	Area where kernel initialization flag of RI850V4 is allocated.
.kernel_const_trace.const	Area where static data of trace function is allocated.
.kernel_data_trace.bss	Area where dynamic data of trace function is allocated.
.kernel_work	Area where system stack, task stack, data queue, fixed-sized memory pool and variable-sized memory pool is allocated.
.*sec_nam*（user-defined area）	Area where task stack, data queue, fixed-sized memory pool or variable-sized memory pool is allocated.

B.1.1 .kernel_system

The size of ".kernel_system" section depends on the trace mode and the kernel library used.
There are four types of kernel library as follows.

- CC-RH version, not supporting the FPU
- CC-RH version, supporting the FPU
- CCV850 version, not supporting the FPU
- CCV850 version, supporting the FPU

The type of the trace mode is selected in the Property panel -> [Task Analyzer] tab -> [Trace] category -> [Selection of trace mode].

1) CC-RH version, not supporting the FPU

Type of the Trace Mode	Size of the Memory Area
Not tracing	20.0K Bytes
Taking in trace chart by hardware trace mode	20.3K Bytes
Taking in trace chart by software trace mode	20.8K Bytes
Taking in long-statistics by software trace mode	20.7K Bytes

2) CC-RH version, supporting the FPU

Type of the Trace Mode	Size of the Memory Area
Not tracing	20.3K Bytes
Taking in trace chart by hardware trace mode	20.6K Bytes
Taking in trace chart by software trace mode	21.1K Bytes
Taking in long-statistics by software trace mode	20.9K Bytes

3) CCV850 version, not supporting the FPU

Type of the Trace Mode	Size of the Memory Area
Not tracing	20.0K Bytes
Taking in trace chart by hardware trace mode	20.3K Bytes
Taking in trace chart by software trace mode	20.8K Bytes
Taking in long-statistics by software trace mode	20.6K Bytes

4) CCV850 version, supporting the FPU

Type of the Trace Mode	Size of the Memory Area
Not tracing	20.2K Bytes
Taking in trace chart by hardware trace mode	20.5K Bytes
Taking in trace chart by software trace mode	21.0K Bytes
Taking in long-statistics by software trace mode	20.9K Bytes

Note The above values are maximum, when using all service calls provided by RI850V4. The value fluctuate corresponding to the type of service calls using.

B.1.2 .kernel_const

The size of ".kernel_const" section depends on the number of information items defined (such as Memory area information and Task information) and the details of the definitions.

The following shows an expression required for estimation ".kernel_const" section size.

In the expression, "align4 (x)" means the result of aligning the value "x" to a 4-byte boundary.

```
KERNEL_CONST =
    224
    + 8 * MEM_AREA_num
    + 24 * CRE_TSK_num
    + 8 * CRE_SEM_num
    + 8 * CRE_FLG_num
    + 8 * CRE_DTQ_num
    + 4 * CRE_MBX_num
    + align4 (2 * CRE_MTX_num)
    + 12 * CRE_MPF_num
    + 12 * CRE_MPL_num
    + 20 * CRE_CYC_num
    + 8 * DEF_INH_num
    + 8 * DEF_SVC_num
    + 12 * ATT_INI_num
    + 8 * VATT_IDL_num
    + align4 (maxint)
    + align4 (TA_ACT_num)
    + align4 (TA_STA_num)
```

Note The keyword in the expression means as follows.

Keywords	Meaning
MEM_AREA_*num*	The number of the definition of the Memory area information.
CRE_TSK_*num*	The number of the definition of the Task information.
CRE_SEM_*num*	The number of the definition of the Semaphore information.
CRE_FLG_*num*	The number of the definition of the Eventflag information.
CRE_DTQ_*num*	The number of the definition of the Data queue information.
CRE_MBX_*num*	The number of the definition of the Mailbox information.
CRE_MTX_*num*	The number of the definition of the Mutex information.
CRE_MPF_*num*	The number of the definition of the Fixed-sized memory pool information.
CRE_MPL_*num*	The number of the definition of the Variable-sized memory pool information.
CRE_CYC_*num*	The number of the definition of the Cyclic handler information.
DEF_INH_*num*	The number of the definition of the Interrupt handler information.
DEF_SVC_*num*	The number of the definition of the Extended service call routine information.
ATT_INI_*num*	The number of the definition of the Initialization routine information.
VATT_IDL_*num*	The number of the definition of the Idle routine information.
maxint	The value defined in the Maximum number of interrupt handlers: maxint; Maximum value of exception code: maxintno.
TA_ACT_*num*	The number of the definition of "TA_ACT" to initial activation state of Attribute: tskatr.
TA_STA_*num*	The number of the definition of "TA_STA" to initial activation state of Attribute: cycatr

B.1.3 .kernel_data

The size of ".kernel_data" section depends on the number of information items defined (such as Task information and Semaphore information) and the details of the definitions.

The following shows an expression required for estimation ".kernel_data" section size.

In the expression, "align4 (x)" means the result of aligning the value "x" to a 4-byte boundary.

```
KERNEL_DATA =
    68
    + 32 * CRE_TSK_num
    + 8 * CRE_SEM_num
    + 8 * CRE_FLG_num
    + 8 * CRE_DTQ_num
    + 12 * CRE_MBX_num
    + 8 * CRE_MTX_num
    + 8 * CRE_MPF_num
    + 8 * CRE_MPL_num
    + 8 * CRE_CYC_num
    + align4 (maxtpri)
```

Note The keyword in the expression means as follows.

Keywords	Meaning
CRE_TSK_*num*	The number of the definition of the Task information.
CRE_SEM_*num*	The number of the definition of the Semaphore information.
CRE_FLG_*num*	The number of the definition of the Eventflag information.
CRE_DTQ_*num*	The number of the definition of the Data queue information.
CRE_MBX_*num*	The number of the definition of the Mailbox information.
CRE_MTX_*num*	The number of the definition of the Mutex information.
CRE_MPF_*num*	The number of the definition of the Fixed-sized memory pool information.
CRE_MPL_*num*	The number of the definition of the Variable-sized memory pool information.
CRE_CYC_*num*	The number of the definition of the Cyclic handler information.
maxtpri	The value of the definition in the Maximum priority: maxtpri.

B.1.4 .kernel_data_init

The size of ".kernel_data_init" section is 4 bytes.

B.1.5 .kernel_const_trace.const

The size of ".kernel_const_trace.const" section depends on the trace mode; that is, the mode selected in the Property panel -> [Task Analyzer] tab -> [Trace] category -> [Selection of trace mode].

In the following table, "align4 (x)" means the result of aligning the value "x" to a 4-byte boundary.

Type of Trace Mode	Size of the Memory Area
Not tracing	align4 (5) Bytes
Taking in trace chart by hardware trace mode	align4 (61) Bytes
Taking in trace chart by software trace mode	align4 (74) Bytes
Taking in long-statistics by software trace mode	align4 (70) Bytes

B.1.6 .kernel_data_trace.bss

The size of ".kernel_data_trace.bss" section depends on the trace mode. The type of trace mode is selected in the "Property panel -> [Task Analyzer] tab -> [Trace] category -> [Selection of trace mode]".

1) Not tracing
The size of ".kernel_data_trace.bss" section is 0 bytes.

2) Taking in trace chart by hardware trace mode
The size of ".kernel_data_trace.bss" section is 4 bytes.

3) Taking in trace chart by software trace mode
The size of ".kernel_data_trace.bss" section depends on the definition in the Property panel -> [Task Analyzer] tab -> [Trace] category -> [Buffer size].
The following shows an expression required for estimating ".kernel_data_trace.bss" section size.
In the expression, "align4 (x)" means the result of aligning the value "x" to a 4-byte boundary.

```
KERNEL_DATA_TRACE.BSS =
    24
    + align4 (TRC_BUF_size)
```

Note The keyword in the expression means as follows.

Keywords	Meaning
TRC_BUF_*size*	The number of the definition of the Property panel -> [Task Analyzer] tab -> [Trace] category -> [Buffer size].

4) Taking in long-statistics by software trace mode
The size of ".kernel_data_trace.bss" section depends on the number of task information items defined and the details of the basic information definitions.
The following shows an expression required for estimation ".kernel_data_trace.bss" section size.
In the expression, "align4 (x)" means the result of aligning the value "x" to a 4-byte boundary.

```
KERNEL_DATA_TRACE.BSS =
    24
    + 8 * CRE_TSK_num
    + align4 (10 *  (intlvl - maxintpri))
    + 8 * maxint
```

Note The keyword in the expression means as follows.

Keywords	Meaning
CRE_TSK_*num*	The number of the definition of the Task information.
intlvl	The value of interrupt level provided in the target CPU.
maxintpri	The value of the definition in the Maximum interrupt priority: maxintpri
maxint	The value of the definition in the Maximum number of interrupt handlers: maxint; Maximum value of exception code: maxintno.

B.1.7 .kernel_work

The size of ".kernel_work" section depends on the information such as Basic information and Task information and so on.

The following shows an expression required for estimating ".kernel_work" section size.

In the expression, "align4 (x)" means the result of aligning the value "x" to a 4-byte boundary.

```
KERNEL_WORK =
    116
    + SYSSTK
    + TSKSTK_total
    + DTQ_total
    + MPF_total
    + MPL_total
```

Note The keyword in the expression means as follows.

Keywords	Meaning
SYSSTK	The value calculated by "System stack".
TSKSTK_*total*	Total amount of the stack size that specified in "Task stack" for each task.
DTQ_*total*	Total amount of the memory size that specified in "Data queue" for each data queue.
MPF_*total*	Total amount of the memory size that specified in "Fixed-sized memory pool" for each fixed-sized memory pool.
MPL_*total*	Total amount of the memory size that specified in "Variable-sized memory pool" for each variable-sized memory pool.

1) System stack

The size of the system stack depends on the details of the Task information and the process of task.

The following shows an expression required for estimation the system stack size required by the RI850V4.

In the following expression, "max(a, b, c)" means the result of selecting the largest value from "a", "b", and "c" (for example, max(1, 2, 3) is 3).

```
SYSSTK =
    max (align4 (INT) + align4 (CYC) , align4 (INI) , align4 (IDL))
```

Note The keyword in the expression means as follows.

Keywords	Meaning
INT	The stack size of the interrupt handler using. If the interrupt handler processes are nested, considers the nest counts. If the interrupt handler is undefined, the stack size of the interrupt handler is nothing.
CYC	The stack size of the cyclic handler using. If multiple cyclic handlers are existed, the maximum value among them. If the cyclic handler is undefined, the stack size of the cyclic handler is nothing.
INI	The stack size of the initialization routine using. If multiple initialization routines are existed, the maximum value among them. If the initialization routine is undefined, the stack size of the initialization routine is nothing.
IDL	The stack size of the idle routine using. If the idle routine is undefined, the stack size of the idle routine is nothing.

2) Task stack

The size of the task stack depends on the details of Task information definitions and processing to be done in the tasks.
The following shows an expression required for estimating the task stack size required by each task defined in the Task information.

```
TSKSTK =
    ctxsz
    + stksz
```

Note The keyword in the expression means as follows.

Keywords	Meaning
ctxsz	This value is determined as shown in Table B-2 according to the target C compiler type and device type specified in the activation options for the CONFIGURATOR CF850V4 (see "18.2.1 Activating from command line" for details of activation options) and the preempt acceptance state specified in the Attribute: tskatr (such as coding language and initial activation state).
stksz	The value corresponding to task processing (defined the value of the Task stack size: stksz, memory area name: sec_nam).

Table B-2 Value of *ctxsz*

Section Name	FPU	Enable Preempt	Disable Preempt
CC-RH version	not supporting the FPU	132	88
	supporting the FPU	136	92
CCV850 version	not supporting the FPU	128	84
	supporting the FPU	132	88

3) Data queue

The size of the data queues depends on the details of Data queue information definitions.
The following shows an expression required for estimating the data queue size by each data queue defined in the Data queue information.

```
DTQ =
    4 * dtqcnt
```

Note The keyword in the expression means as follows.

Keywords	Meaning
dtqcnt	The value of the definition in the Data count: dtqcnt, memory area name: sec_nam.

4) Fixed-sized memory pool

The size of the fixed-sized memory pools depends on the details of Fixed-sized memory pool information definitions.
The following shows an expression required for estimating the fixed-sized memory pool size required by each fixed-sized memory pool defined in the Fixed-sized memory pool information.
In the expression, "align4 (x)" means the result of aligning the value "x" to a 4-byte boundary.

```
MPF =
    align4 (blksz) * blkcnt
```

Note The keyword in the expression means as follows.

Keywords	Meaning
blksz	The value of the definition in the Basic block size: blksz, memory area name: sec_nam.
blkcnt	The value of the definition in the Block count: blkcnt.

5) Variable-sized memory pool

The size of the variable-sized memory pools depends on the details of Variable-sized memory pool information definitions.
The following shows an expression required for estimating the variable-sized memory pool size required by each variable-sized memory pool defined in the Variable-sized memory pool information.
In the expression, "align4 (x)" means the result of aligning the value "x" to a 4-byte boundary.

```
MPF =
    align4 (blksz) * blkcnt
```

Note The keyword in the expression means as follows.

Keywords	Meaning
mplsz	The value of the definition in the Pool size: mplsz, memory area name: sec_nam.

B.1.8 .sec_nam(user-definied area)

The size of ".sec_nam (user-defined area)" depends on the details of information definitions (such as Task information and Data queue information).

Note This section is necessary when the task stack or data queue area is defined to be allocated outside ".kernel_work" section in the Task information or Data queue information.
Estimate the memory size for this section with reference to the descriptions in "B.1.7 .kernel_work".

APPENDIX C SUPPORT FOR FLOATING-POINT OPERATION COPROCESSOR

The RI850V4 supports the floating-point operation coprocessor of the RH850.

The RI850V4 manipulates the floating-point configuration/status register (FPSR) for floating-point operation. The user can change the floating-point operation settings from processing programs as needed by changing this register value.

The value of FPSR is essentially specified independently for each processing program and is not inherited between processing programs.

However, the RI850V4 does not manipulate FPSR when an extended service call routine starts or ends. For this reason, an extended service call routine inherits the FPSR value from the previous processing executed before the extended service call routine, and the value changed in a processing program is retained after the program ends.

See table Table C-1 for the register value when each processing program is initially activated.

Table C-1 Register Values at Activation of Each Processing Program

Processing Program	Initial FPSR Value
Task	User setting
Cyclic handler	User setting
Interrupt handler	User setting
Extended service call routine	Inherits the value before activation.
Idle routine	User setting

Note 1 If a task is suspended and then resumed, the FPSR is restored to the value before the suspension.

Note 2 "User setting" for FPSR in the above table is the value specified as the FPSR register information in the system configuration file.

Revision Record

Rev.	Date	Description	
		Page	Summary
1.01	Sep 30, 2015	-	First Edition issued

RI850V4 V2 User's Manual:
Coding

Publication Date: Rev.1.01 Sep 30, 2015

Published by: Renesas Electronics Corporation

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HALII Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

© 2015 Renesas Electronics Corporation. All rights reserved.

Colophon 4.0

RI850V4 V2

Renesas Electronics Corporation

R20UT2889EJ0101